Automated Program Transformation Through Proof Transformation

Peter Madden

PhD
University of Edinburgh
1991

Abstract

We investigate program optimization and program adaptation (or *specialization*) by the transformation of (constructive) synthesis proofs. Synthesis proofs which yield inefficient programs are transformed into *analogous* proofs which yield more efficient programs. These proofs are based on a Martin-Löf type theory logic and proved within the OYSTER proof refinement system (Martin-Löf, 1979; Martin-Löf, 1984).[1]

The problems of automated program synthesis and verification have already been addressed within the *proofs as programs paradigm* (Horn & Smaill, 1990; Constable *et al*, 1986), (Bundy *et al*, 1990a). By using *constructive logic*, the task of generating programs is treated as the task of proving a theorem. By performing a proof of a formal specification expressed in constructive logic, stating the *input-output* conditions of the desired program, an algorithm can be routinely extracted from the proof.

We have implemented a system – the *meta-level* OYSTER *proof transformation system* (MOPTS) – for optimizing programs through the transformation of (OYSTER) synthesis proofs. The MOPTS has the desirable properties of *automatability*, *correctness* and various mechanisms for *reducing the transformation search space*, and various *control mechanisms* for guiding search through that space.

A contribution afforded by proof transformations is that, in addition to program synthesis and verification, the problem of program transformation is also tackled by transposing the task to the proofs as programs paradigm. As with synthesis and verification, knowledge of theorem proving, and in particular automatic proof guidance techniques, can be brought to bear on the task. Furthermore, such transformations allow the human synthesizer to produce an elegant *source* proof, without clouding the theorem proving process with efficiency issues, and then to transform this into an opaque proof that yields an efficient *target* program.

---

[1] OYSTER is the Edinburgh Prolog implementation, and extension, of NUPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell (Bundy *et al*, 1990b),(Horn & Smaill, 1990; Constable *et al*, 1986).

To accomplish program transformation *through* proof transformation, we have successfully, and for the first time, adapted a range of program transformation techniques to the proofs as program paradigm, notably: the *tupling* technique for "merging" repeated (sub)computations, (Pettorossi, 1984) (Chin, 1990), and the fold/unfold technique for transforming inefficient functional programs into equivalent, more efficient, functional programs by a process of unfolding and folding definitions (Darlington, 1981a). Throughout the course of this thesis we shall highlight the benefits of our "proofs as programs" approach to transformation, particularly with respect to search, correctness and automatability.

An important property of both program optimization and program specialization, particularly regarding *automation* and *control*, is the fact that information contained in proofs, which may go beyond that needed for simple execution, is exploited for the purposes of transformation: a proof will contain more information than the program which it specifies since a program need contain no more information than that required for execution. A proof, on the other hand, will contain the *thinking behind the program design.* A main contribution of this thesis stem from an investigation into how this extra information can be exploited for the task of program transformation.

A key feature of our approach to program optimization consists in the transformation of the various induction schemas employed in OYSTER synthesis proofs. Of particular importance to inducing recursion in the extracted algorithm is the employment of *mathematical induction* in the synthesis proofs: to each form of induction employed in the proof there corresponds a dual form of recursion. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm.

The specialization (sub)system has been reconstructed from, and is compared with, the original implementation (Goad, 1980b; Goad, 1980a). The purpose of the reconstruction was primarily as a preliminary investigation into program transformation through proof transformation – (Goad, 1980b; Goad, 1980a) being the only other example of a working system that performs such transformations. We

shall, however, discuss certain properties of the reconstruction which, we believe, mark an improvement over Goad's original design.

Table of Contents

List of Figures

Acknowledgements

First and foremost, I would like to express my thanks and gratitude to my supervisor, Alan Bundy, for his tireless and indispensable support, both academic and non-academic, throughout the course of my PhD research.

I would also like to express my thanks and gratitude to the following people: Alan Smaill, my second supervisor, for his many suggestions and patient assistance regarding the more mathematical aspects of this thesis.

Andrew Ireland, who provided numerous useful suggestions and stylistic tips regarding the documentation of my PhD research.

Paul Bryna for general encouragement, and endless help with the Latex Documentation System.

Mitch Harris and Toby Walsh for providing financial assistance, and a roof, during difficult times.

I am also very grateful to my parents, my brother, Tony, and my girlfriend, Heidi, simply for being there in times of need.

This thesis is dedicated to my family and to Heidi

This research is supported by SERC grant GR/D/44270, and a SERC studentship to the author.

Declaration

I declare that this thesis has been wholly composed by myself, and that the research documented is my own.

P. Madden

Formal Publications

Some of the research documented in this thesis has already appeared in print, either as formal publications or as University of Edinburgh, Dep. of Artificial Intelligence Research Papers. We list the formal publications below, and shall refer to these, along with the Reasearch Papers, in the main text.

Madden, 1988a P. Madden. Automatic program optimization via the transformation of NuPRL synthesis proofs. In Clarke, L., (ed.), *Proceedings of the 1988 Alvey Technical Conference.* The Alvey Directorate. 1988. Extended version available as Research paper 392, Dept. of Artificial Intelligence, Edinburgh.

Madden, 1989 P. Madden. The specialization and transformation of constructive existence proofs. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufmann. 1989

Bundy *et al*, **1988** Bundy, A., Sannella, D., Giunchiglia, F., Harmelen, F. Van, Hesketh, J., Madden, P., Smaill, A., Stevens, A. and Wallen, L. Proving properties of logic programs: A progress report. In *1988 Alvey Conference.*

Chapter 1

Introduction: Overview and Aims of Thesis

1.1 Introduction

A problem that pervades the fields of Computer Science and Artificial Intelligence is that the demands for reliability and quality of software often outstrip the available tools. A solution to this problem is offered by the field of *automatic programming*. This can be broken down into three main, interrelated, sub-fields:

- The *automatic generation*, or *synthesis*, of programs from specifications (input-output relations).
- The *automatic verification* that a program meets its specification.
- The *automatic transformation* of one program into a more efficient program meeting the same specification.

So, by tackling these issues, software reliability can be improved, provided that it is easier to write bug-free specifications than bug-free programs. For several years the *Mathematical Reasoning Group*, MRG, in the *Edinburgh University Department of Artificial Intelligence*, have undertaken research, under the direction of Prof. Alan Bundy, into the field of automatic programming (Bundy *et al*, 1988; van Harmelen, 1989; Bundy *et al*, 1990a; Bundy *et al*, 1991). The first two issues above have been tackled within the OYSTER-CLAM proof refinement environment and form part of an ongoing research project:[1] By using *constructive logic*, the

[1] OYSTER is the Edinburgh Prolog implementation, and extension, of NuPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell

task of generating programs is treated as the task of proving a theorem. By performing a proof of a formal specification expressed in constructive logic, stating the *input-output* conditions of the desired program, an algorithm can be routinely extracted from the proof. Knowledge of theorem proving, and in particular automatic proof guidance techniques, are used in this task.

Of particular importance to inducing recursion in the extracted algorithm is the employment of *mathematical induction* in the synthesis proofs: to each form of induction employed in the proof there corresponds a dual form of recursion. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm.

The third issue, that of program transformation, is the latest to be tackled by the MRG and forms the main subject of this thesis: *the automatic transformation of programs by transforming their synthesis proofs.*

Transformation systems may have many applications, although in this thesis we shall concentrate on two applications, *specialization* and *recursive program optimization* (although examples of some of the other applications are subsumed by these two). The main form that the OYSTER transformations will take are:

1. OPTIMIZATION: The optimization of *recursive programs* by transforming the corresponding *inductive synthesis proofs.*

2. SPECIALIZATION: The adaptation, or *specialization*, of programs to special situations by partially evaluating and then simplifying synthesis proofs through proof tree *pruning* transformations.

In the remainder of this chapter we give an overview of the thesis research.

1.1.1 Program Synthesis in a Constructive Logic

In keeping with the introductory nature of this chapter, the following exposition of program synthesis through constructive theorem proving will be brief. We shall give an extensive account of the system in *Chapter 2.*

(Horn & Smaill, 1990; Constable *et al*, 1986). The OYSTER-CLAM system is an extension of OYSTER which is designed to automatically construct formal *reasoning patterns*, or *proof-plans* which can then be used to guide the synthesis proofs (*cf. Chapter 2*).

If we represent the program specification as **specification**(input,output), then by finding a constructive proof of:

$$\vdash \forall \text{ inputs } \exists \text{ output } \textbf{specification}(\text{input, output})$$

we can *extract* an algorithm, **alg** such that,

$$\vdash \forall \text{ input } \textbf{specification}(\text{inputs, } \textbf{alg}(\text{input})).$$

alg is known as the *extract term* (or *extract algorithm*) of the constructive proof.

So, for example, suppose we wish to compute a value for the integer log to the base 2 of our input, then from a proof of the following specification:[2]

$$\vdash \forall\, inputs : integer\ \exists\, output : integer\ (\ 2^{output} \leq input\ \&\ input < 2^{output+1}\ )$$

we extract an algorithm *alg* which satisfies the following:

$$\vdash \forall\, input\ (\ 2^{alg(input)} \leq input\ \&\ input < 2^{alg(input)+1}\ )$$

and which does the required job. Proving that some given extract algorithm does satisfy the above is known as *verification.*

Martin-Löf type theory is an intuitionistic, constructive, higher order, typed logic (Martin-Löf, 1979; Martin-Löf, 1984). It is especially suitable for the task of program synthesis, since executable code is built up as a proof is constructed such that all elements of the program correspond to elements of the proof (the converse is *not* true, proofs contain additional information, *cf. chapter 2*). In other words, each rule of inference has an associated rule of program construction. The constructive properties of the logic mean that we avoid the possibility of a pure existence proof in which the existence of an output is proved without any implicit algorithm to construct the object being defined.

Bundy succinctly expresses why it is important that the logic employed for OYSTER *synthesis* be *constructive* (Bundy, 1988a):

> ...when a theorem is proved by considering a number of cases, it is constructively necessary to be able to tell the cases apart: if we are in such a position, the resulting algorithm will be able to appeal to some test as to which case is relevant. In classical logic, there may be no such test, so a proof that uses (classical) axioms like
>
> $$\phi \vee \neg\phi$$

---

[2]Typing is not, of course, restricted to integers. Types can be natural numbers, lists of natural numbers (or integers), sets and so forth (*cf.* §2.2.1).

in a proof by cases, where there is no decision procedure for ϕ, will fail to yield an algorithm in the way described.

This example shows why the use of constructive logic for synthesing *recursive* algorithms is particularly beneficial, since all recursive program syntheses require inductive proofs, and all inductive proofs are proofs by cases.

In *Chapter 2* we shall describe the specifics of an interactive constructive proof editor, OYSTER.

1.2 Aims and Contributions of the Thesis

The system described in this thesis is to be viewed as a research tool for the development of proof transformation methodologies in that the implemented system provides some basic, although novel, *proof* transformation techniques which can be expanded upon in the course of future research.[3] Apart from the specialization system described in (Goad, 1980b; Goad, 1980a), this thesis reports on the only other working transformation system that optimizes programs through proof transformations (excluding the authors previous publications concerning program through proof transformation (Madden, 1991; Madden, 1988b; Madden, 1988a; Madden, 1989). The applications of the transformations are considerably broader than those documented in (Goad, 1980a).

The main question which this thesis addresses is how program synthesis proofs can be exploited in order to transform programs *and* what advantages this approach has over the more traditional approach to program transformation where transformation rules are applied directly to the *source* code in order to construct

[3]The implemented techniques are discussed throughout this thesis, in particular in *Chapters 4* and *5*. The merits and contributions of the proof transformation approach to program modification are summarized in §6.1 through §6.2. Some further avenues for research, that build upon the implemented system, are discussed in §6.3.

the *target* program (*Chapters 5* and *6*).[4] We also, as a precursor to the aforementioned issue, investigate how the *efficiency* of a synthesized program is dependent upon the way the synthesizer employs proof constructs in order to satisfy the program specification (*Chapter 3*).

In general, whether referring to proof or program specifications, the demands for efficiency of programs are succinctly expressed by quoting from (Bjerner, 1989) (*italics* added by the author):

> The first criterion on which a program is judged is the correctness with respect to its specification. The second criterion is the efficiency of the program with respect to other programs *satisfying the same specification*, which is reflected by time and space complexity of the program.

and the problem with satisfying such demands are expressed in the following quote from (Feather, 1979b) which highlights the trade off between the efficiency of programs versus their modifiability:

> ...an alarming proportion of programming effort is devoted to maintaining or modifying existing software as opposed to creating new products. This process of modification is very poorly understood...the root cause of this trouble is the way the programming task is approached; the code available for modification is code meant to be run. Even if this has been written in a high level language it has to be written to be efficient at the expense of clarity and therefore modifiability.

The efficiency issue is addressed by describing program optimization *through* OYSTER proof transformation.

We should mention that, regarding the first quotation above, some researchers would regard to what degree the program meets the users requirements as a primary criteria, although they would no doubt agree that correctness and efficiency are important secondary and tertiary requirements. Regarding the second quote it

---

[4]The terms *source* and *target* are used throughout this proposal and are independent of any domain. The former simply refers to a past solved problem and the latter to that current problem for which the proof plan construction is attempted by transforming the source proof plan.

would perhaps be more accurate to say that whereas a great deal of programming effort *is* devoted to modifying existing software, the process lacks any uniform methodology.[5]

Human theorem provers are usually trained to find short, elegant proofs rather than long opaque ones, despite the fact that the latter may yield more efficient programs. Hence, proof transformations allow the human theorem prover to produce an elegant *source* proof, without clouding the *design process* with efficiency issues, and then to transform this into an opaque proof that yields an efficient *target* program.[6]

Such transformations will require a general knowledge of the relationships between *inductive proof constructs* and the extract *recursive program constructs*, and a knowledge of the (relative) efficiency between the various *recursive program constructs.*

1.2.1 Desirable Properties of the Proof Transformation System

The author has implemented a program specialization and optimization system. Although the system should be regarded as in its embryonic form it offers one of the first system designs for program optimization through proof transformation, and is the first such design to be implemented (to the authors knowledge).

The main advantages of *proof* transformation over the more traditional source to target transformations, which transform program constructs *directly*, stem from

---

[5]Thanks to Dr. Tim Smithers for pointing out the somewhat contentious nature of these quotes.

[6]Within the context of program syntheses from formal specifications, the term *design*, or *design process*, is taken to mean the cognitive processes involved in selecting one particular proof (and all the sub-proofs within) of a specification, as opposed to refining any of the other potentially infinite number of proof trees that would satisfy the same specification. Examples of deifferent proof trees satisfying the same specification, but involving different designs, are provided in *Chapters 2, 4* and *5.*

the properties of the proof transformation system (which may in turn be due to the properties of the constructive theorem proving). The most important properties of the proof transformation system are outlined below:

- *Program Through Proof Transformation*: Programs are automatically transformed by transforming synthesis proofs:
  - Programs are adapted to special situations, *specialization*, by the *partial evaluation* of synthesis proofs followed by special *pruning* transformations.
  - Recursive programs are optimized by transforming the induction schemata employed in the corresponding synthesis proofs from which they are extracted.

- *Automatability*: The system is *fully automatic* with regard to the source to target transformations: the source proof is automatically transformed to a target proof, from which an extract program is automatically (and trivially) extracted.

- *Correctness*: With many of the non-theorem proving approaches to transformation there is no obvious *correctness checking procedure* that the transformations have preserved the source specification. In the case of proof transformation it is a simple task to check, after *each* transformation if desired, that the resultant proof still satisfies the source proof *specification.*

- *Exploiting Non-Algorithmic Information*: A proof will contain more information than the program which it specifies since the program need contain no more information than that required for execution: proofs represent a *program design record* because they encapsulate the reasoning behind the program design by making explicit the procedural commitments and decisions made by the synthesizer.

  That is, proofs of program specifications differ from straightforward programs in that more information is formalized in the proof than in the program:

- A description of the task being performed;
- a verification of the method;
- an account of the dependencies between facts involved in the computation;

and hence that proofs lend themselves better to *transformation* than programs since one expects that the data relevant to the transformation of algorithms will be different and more extensive than the data needed for simple execution (this point is expounded upon in much greater detail throughout *Chapters 4* and *5*.

- *Reducing Search Space/Increasing Control*: The aforementioned information, contained in proofs but not programs, is exploited by the proof transformation system to both reduce the search space associated with the transformations (the *transformation space*), and for controlling the search through the transformation space. We shall see, in *Chapters* 4 and 5, that much of the analysis and search associated with certain program transformation techniques, reviewed in *Chapter 3*, is reduced when these techniques are transposed to the *proofs as programs* paradigm.

- *Termination*: With the non-theorem proving approaches to transformation there is no obvious stopping condition, one simply hopes that an executable program, and one that does the desired job, *will* be achieved. Whereas in the case of proof transformation, the termination point of a transformation corresponds to a completed target proof that satisfies the target specification (and in the case of optimization, but not necessarily specialization, the target proof will satisfy the *same* specification of the source).

- *Meta-level Control*: The proof transformation techniques are expressed in terms of transformation *tactics*, with pre- and post- conditions, which oper-

ate on the object-level OYSTER proofs.[7] By performing transformations at the meta-level, according to whether or not the tactic conditions are satisfied, we considerably reduce the amount of search associated with constructing the target proof at the object-level.

- *The Proof-Program Uniformity*: For each transformation operation performed on a synthesis proof there will be a corresponding transformation in the program. This property means that synthesis and transformation can be treated uniformly, henceforth referred to as the *proof-program uniformity.*

  This differs from the case where transformation operators are applied to the source program itself. In such a case transformation and synthesis cannot be treated uniformly, and in practice the resulting combination of both processes leads to an increase in complexity.[8]

- *Generality*: The majority of constructive proofs that employ *mathematical induction* share a common structural framework: there is a high degree of similarity in the overall shape of the inductive *proof trees* (and in the strategy employed in inductive proofs).[9] This provides the potential to build *general* proof transformation strategies which operate on a wide class of inductive proofs and, thereby, can optimize a wide class of recursive programs.

- *Exploiting Theorem Proving Techniques*: As stated in the introductory paragraph to this chapter, knowledge of theorem proving, and in particular auto-

---

[7]The terms *object* and *meta* are, of course, relative terms and the transformation system is referred to as a *meta-level* system since the *transformation operators*, or *rules*, act upon the (sub)proofs of the synthesis system. Hence we refer to the OYSTER refinement proofs as belonging to the *object-level* synthesis system.

[8]It can be argued that since proofs are more complex than the resultant extracts, therefore, in some cases, it may be easier to transform the extracts directly. This is, however, to miss the point of proof transformation: the extra complexity in the proof may be due precisely to information that we wish to exploit to guide transformation.

[9]The automatic CLAM proof-planning system *formally* encapsulates this common shape of inductive proofs in a *meta-logic*. The proof plans are created automatically and can then be used, as a general proof pattern to guide the refinement of *specific* specifications (Bundy *et al*, 1991; Bundy *et al*, 1990b).

matic proof guidance techniques, are exploited for the task of program generation through constructive theorem proving. A knowledge of such proof guidance techniques are embodied in the proof transformation strategies since it is by abstracting, and if necessary adapting, the means by which a source proof is constructed that the target proof is developed.

- *Reacting to Changing Specifications*: It can often be problematic to react to changing specifications when programs are defined through proofs due to the difficulty to make local changes to a proof structure. The following quote from (Pfenning, 1988) addresses this problem:

  > Proof transformations provide a way of making those local changes and propagating them throughout the proof. This process very clearly identifies places where additional theorem proving is required in order to meet the changed specification or verify the modified program. This is in sharp contrast to traditional programming where maintenance under changing requirements is one of the most difficult and costly phases in the software life-cycle and relies entirely on the programmer's understanding of the code he has to modify.

The specialization of a program through the partial evaluation of the initial specification, and corresponding synthesis proof, provides a good example of how proof transformation can assist in modifying a programs internal structure in accordance with an initial modification of its specification.

1.3 Overview of Thesis

In this section we give a brief outline of the main functions of the proof transformation system described in this thesis. A description of the proof transformations in terms of the properties of constructive theorem proving is provided in *Chapter 2*. The details of the techniques and methodologies employed by the proof transformation system, along with worked examples, are provided in the relevant chapters (*Chapters 4 and 5*).

The optimization of recursive programs and the specialization of programs are best thought of as two applications of the same *meta-level* OYSTER *proof transformation system* (MOPTS), since both applications share the same central mechanisms. However, were appropriate, we shall distinguish the specialization application from the optimization application by referring to the former as the *proof specialization (sub)system*, or PSS, and referring to the later as the *inductive proof optimization (sub)system*, or IPOS.

1.3.1 Synthesizing Algorithms From a Common Specification: *Interactive Proof Transformation*

In *Chapter 2* we investigate how *different* object-level proof syntheses, refined from the *same* specification, can yield algorithms which are equivalent in functionality but differ considerably in terms of computational efficiency.

We concentrate on how, during constructive proof refinement, the employment of various *induction schemata* induces different *recursive behaviour* in the algorithms extracted from the synthesis proofs. This will give us information concerning the *recursive efficiency* of the various program constructs induced by the inductive proof constructs, and will therefore prove useful in the design of the MOPTS *and also* in measuring the performance of the source to target optimizations.

Fig. 1–1 schematically depicts the multi-syntheses of various *recursive* algorithms, Ω_n, from a *single* proof specification S, where

$$S = \forall \text{ inputs } \exists \text{ output } \mathbf{specification}(\text{input, output}).$$

Each downward arrow represents a completed synthesis, and each synthesis employs a different *induction schema*, Ind_n. The *object-level* syntheses of source and target algorithms from a common specification will also serve to explain our system.

In particular, the syntheses will provide us with information concerning how the recursive *behaviour* induced in the extract program is controlled by the par-

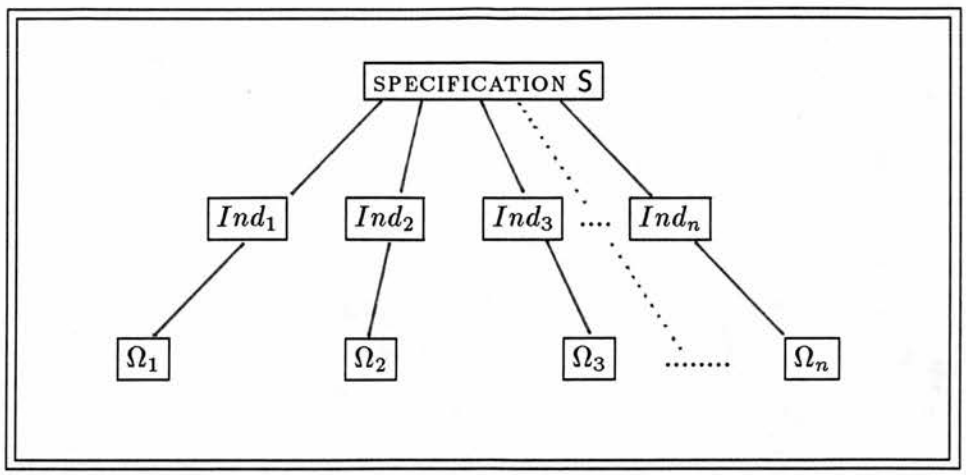


Figure 1–1: Multiple inductive syntheses from common specification.

ticular induction scheme chosen, and how the induction cases are subsequently instantiated.

Furthermore, the *object-level* syntheses serve as a useful source of comparison with the more traditional program transformation systems which by a process of multiple refinement application, produce a variety of algorithms from a *single* source *specification.* Many of these systems, like the OYSTER system, are interactive.

1.3.2 Optimization of Recursive Algorithms By Transforming Inductive Proofs

The computational efficiency of a recursive algorithm is directly related to the *form* of the recursion. The way in which an algorithm recurses on its input can be *controlled* by the way in which mathematical induction is employed in the algorithm's synthesis.

Boyer and Moore have done extensive work on heuristics for inductive proofs (Boyer & Moore, 1979; Boyer & Moore, 1988). Relationships between induction and recursion have been generalized such that most recursive structures have a corresponding induction schema which can be employed to synthesise programs exhibiting the desired recursive behaviour (Stevens, 1988).

The crucial element in the transformation is that *recursive programs are optimized by transforming the induction schema employed within the corresponding synthesis proofs.* This is a novel approach to program optimization.

Fig. 1–2 schematically depicts the source to target meta-level transformation: a single source inductive synthesis proof is depicted on the *left hand side* of the diagram: The proof yields a complex source algorithm, *exp*, which recurses with *exponential* behaviour due to the fact that a particular induction – *course_of_values* – is employed during the synthesis (the term *ext.* represents the program extraction process).

The target proof is represented on the *right hand side*: the proof is not interactively refined from the specification, as is the source, but rather *automatically* constructed by the application of operators which map and then transform portions of the source proof. In particular, the source *course_of_values* induction is transformed into the more efficient *stepwise* target induction, thus yielding a target extract algorithm that recurses on its data-structure in more efficient *linear* fashion.

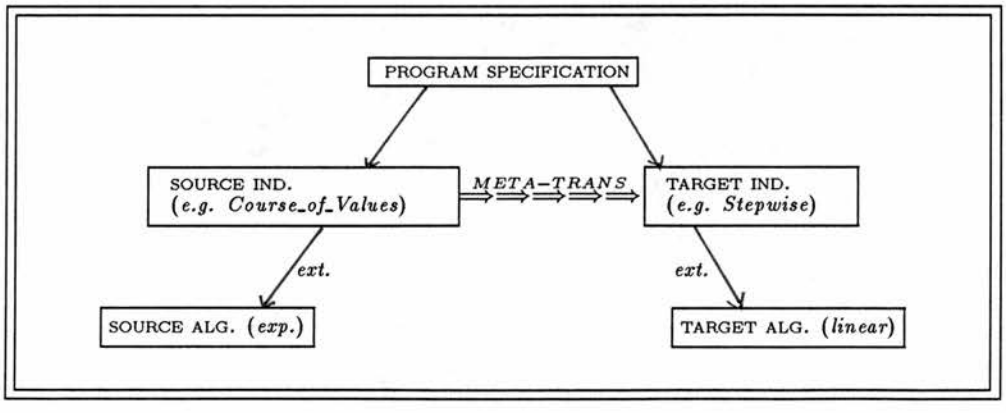


Figure 1–2: Recursive program optimization through induction schema transformation.

A Brief Example of Transformation of Induction Schemas: Linearization of *Fibonacci*

We can construct at least two proofs, within OYSTER from which two alternative recursive algorithms can be extracted, each of which computes the *Fibonacci* function. The difference between the two syntheses is that each employs a different induction schemata: *course_of_values* induction will induce *course_of_values* recursion in the *Fibonacci* extract algorithm and *stepwise* induction will induce *stepwise* recursion. The technical descriptions of these two types of induction will be provided in §2.2.3. For the present we present an example of a *course_of_values* function definition and a *stepwise* function definition (each of which can be synthesized using the respective induction schema).

To employ *course_of_values* induction in the synthesis of an algorithm which takes as input n requires appealing to all, or a subset of, the output values obtained when the input is any value less than n.[10] Using a standard functional notation, we can represent an algorithm which computes the *Fibonacci function and* which is extracted from a course_of_values proof as follows:

source definition:

$$\begin{aligned} fib(0) &= 1; \\ fib(1) &= 1; \\ fib(n+2) &= fib(n+1) + fib(n). \end{aligned}$$

By employing *course_of_values* induction we obtain an algorithm such that in order to calculate $fib(n)$ one must first calculate $fib(n-1)$ and $fib(n-2)$. Each of these sub-goals leads to another two recursive calls on fib and so on. In short, the *recursive branching rate* is 2, and the computational tree is exponential where the number of recursive calls on fib approaches 2^n.

However, by employing the *tupling technique* for *linearization*, we can *automatically* transform the course_of_values inductive proof into a target proof that,

---

[10]The completed proofs are displayed, and examined, in Chapters 2 and 6.

in effect, employs stepwise induction. In general, the tupling technique transformations allow for the "collapsing" of less efficient induction schemas into more efficient ones. In this way subsidiary recursive calls, associated with the source induction schemas, can be merged into a single step in the optimized definition. This means we do not need to do the *object-level* synthesis of the more efficient *stepwise recursive* target program. The tupling technique is explained, within the context of program transformation, in §3.2.1, and, within the context of the author's proof transformations in *Chapter 5* (specifically §5.2).

The rationale embodied in the tupling technique is *similar* to that described in (Burstall & Darlington, 1977b), (Darlington, 1981a), and later in (Chin, 1990): A function, g, is constructed which combines the values of the two subsidiary recursive calls of the less efficient course_of_values definition (the function $\langle x, y \rangle$ can loosely be interpreted as a variadic function which simulates the action of tuples). The process results in a proof that yields a function definition corresponding to the following:

target definition:

$$\begin{aligned} fib(n) &= m \quad \textit{where } \langle \_, m \rangle = g(n); \\ g(0) &= \langle 1, 1 \rangle; \\ g(n+1) &= \langle u1 + u2, u1 \rangle \textit{ where } \langle u1, u2 \rangle = g(n). \end{aligned}$$

In this case there is no recourse to the original fib definition and $g(n)$ requires only n recursive calls (*stepping down* to the base case $g(0)$). In other words, the computational tree resulting from stepwise induction is *linear*, with a branching rate of 1, and hence the resulting algorithm requires far less computational effort in computing $fib(n)$ than that synthesized by employing *course_of_values* induction.

Fig. 1–3 and **fig.** 1–4 depict differences in the computational trees for $fib(5)$ using respectively course-of-values induction and stepwise induction. Note especially the redundant (repeated) nodes in the tree for course-of-values induction. Again, the angled brackets in the stepwise sequence symbolize *tuple formation* in that the output of each recursive pass is some function of the arguments within the brackets. This thesis offers the first instance of the tupling technique being

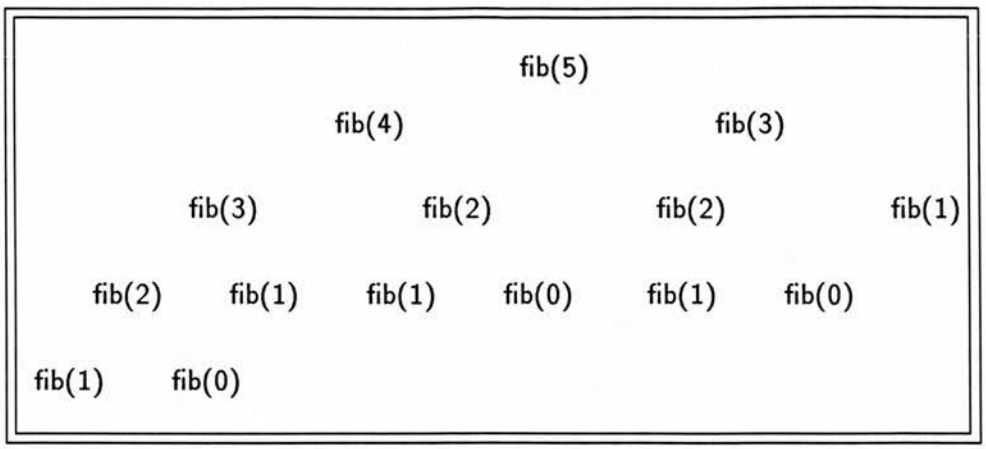


Figure 1–3: Computational tree for $fib(5)$ induced by *course_of_values* induction

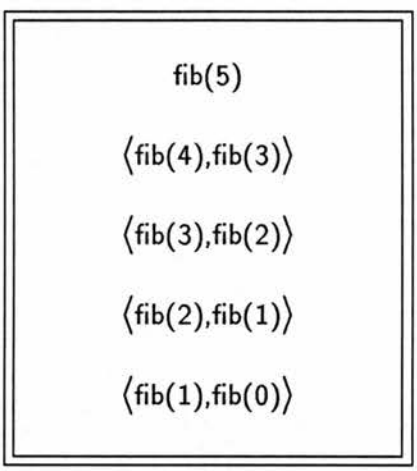


Figure 1–4: Computational tree for $fib(5)$ induced by *stepwise* induction

employed within the context of proof transformation.

The use of tupling for (i) interactively refining the more efficient stepwise algorithm from a single specification and (ii) automatically optimizing the course_of values algorithm, are discussed in *Chapters 4 and 6* respectively.

The general strategy of program transformation employed in (Darlington, 1981a) and (Chin, 1990) originated from (Burstall & Darlington, 1977b) and is referred to as the *fold/unfold* strategy. It basically consists in defining the target program in terms of the source, and then, by a process re-writing recursive definitions, deriving a recursive definition for the target program which is independent of the

source definition. This general strategy has since been incorporated, in a variety of guises and applications, in many program transformation systems (*cf. Chapter 3*). The two most problematic steps in the fold/unfold strategy are:

(i) obtaining the initial definition of the target in terms of the source; and

(ii) the control problems associated with when to apply the re-writing step(s) which eliminate any reference to the source definition from the target recursive step.

For a specified class of program, the MOPTS tackles both thesef problems with considerable success.

1.4 Specialization of Programs Through Partial Evaluation and Pruning of Proofs

The second main task of the MOPTS is program adaptation, or *specialization*, performed by the PSS. As with recursive program optimization, specialization exploits information contained in proofs, which goes beyond that needed for simple execution. The system employs many of the same basic transformation operators as the IPOS.

The specialization work is based upon Goad's specializtion system, (Goad, 1980b; Goad, 1980a), one of only two other researchers as far as the author is aware, who are also involved with modifying programs through proof modification.[11] Goad's system has been successfully reconstructed, *and* extended, by the author in the OYSTER *proof refinement* environment and subjected to test on a number of examples.

---

[11]The other being (Pfenning, 1988) which, although it post-dates the author's earlier work concerning proof transformation, (Madden, 1986; Madden, 1988d; Madden, 1988b), discusses a *system design* for program transformation through proof transformation (*cf. Chapter 3*).

There are various ways of explaining specialization, depending on whether one is taking a theoretical or more practical stance. We shall cover the alternatives throughout the course of this thesis (especially in *Chapter 5*).

For the present, specialization is best described as a technique for *adapting* programs to *specific situations* (viz. specific classes of inputs) by *partially evaluating* their corresponding synthesis *proofs* (for example, adapting a sorting algorithm or a bin-packing algorithm to operate, with maximum efficiency, on input of a specific length). An important qualitative difference between the PSS and the IPOS applications is that whereas the optimization transformations preserve both the functionality and specification of the source program, the specialization transformations may alter the functionality of the algorithm *without necessarily altering the proof specification.*

As in (Goad, 1980a), the specialization process has three distinct stages: partial evaluation (*PE*), normalization (*NORM*) and dependency pruning (*DEP PRUNE*) corresponding, respectively, to the three (sub)transformation headings in **fig.** 1–5 below: Synthesis proofs explicitly contain *dependency* information which is im-

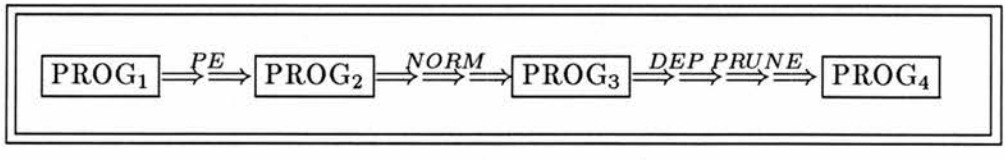


Figure 1–5: Schematic view of specialization

plicit, or usually absent, in programs. Each proof node contains a record of assumptions, hypotheses, and possibly lemmas, used thus far in the proof, and refinements can explicitly appeal to each of these. As such, any refinement which invokes an associated program construction rule may appeal to earlier proof constructs associated with other proof constructs. In this way we can regard synthesis proofs as incorporating *dependency graphs* in that the inter-dependencies between procedural commitments and decisions made by the user are explicitly represented, and thus subject to inspection and modification.

Partial evaluation of a constructive existence proof produces a rough equivalent to a *grounded dependency graph* from which dependency, and hence redundancy, information can be gleaned. This information then drives the pruning mechanisms. The treatment of the partially evaluated proofs as grounded dependency graphs is the author's idea and facilitates the pruning procedures.[12]

Normalization is designed to remove,

(i) identical (or equivalent) sub-computations, and,

(ii) remove unsatisfied(false) case split branches

from programs by exploiting redundancy information explicitly exhibited in the synthesis proof from which the source program was extracted.

Dependency pruning, on the other hand, exploits the remaining redundancy information in the *specialized - partially evaluated* - proof in order to remove redundant, but not repetitive, sub-computations.

Each of the three sub-applications (partial evaluation, normalization, and dependency pruning) are treated as practical applications in their own right. In particular, we shall concentrate on the partial evaluation of OYSTER *induction schemata.*

As with the IPOS of the MOPTS, all source to target transformations performed by the PSS satisfy the desirable criteria for a transformation system of *correctness*, *generality*, and *automatability.*

The interesting features of the PSS derives from the properties of the (object-level) OYSTER proofs and the means by which such proofs are transformed. Usual program transformations do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour. In the case of specialization through proof transformation, however, we are able to show how

---

[12]We shall see that this also brings the *specialization* and *pruning* system into the domain of explanation based learning transformation techniques (*cf. Chapter 3*).

a target program that computes a *different* function to the source program can be used to provide the *same* output as the source for a particular assignment to one of the parameters (i.e a particular instantiation of one of the specification input variables).

There are notable differences in methodology between Goad's system and the reconstruction (PSS). In particular, the means by which the PSS ensures the correctness of the pruning transformations requires no additional "validity proofs" in order to establish that the pruning operations preserve the validity of the target proof with respect to it's specification. This is due to the nature of the proof abstractions that the PSS (and the MOPTS in general) use in order to perform transformations. A further difference, also due to the properties of the MOPTS proof abstractions, is the open-ended nature of the PPS design: all transformations terminate with, like Goad's system, a target program, and, unlike Goad's system, a complete target proof. This allows for the possibility of further proof transformations to follow the initial specialization. Details of these and other differences will be covered in subsequent chapters.

1.5 Summary

To recap, program synthesis amounts to the finding of a formal proof, in *constructive logic*, that a *program specification* can be satisfied. Such proofs yield programs which will perform the specified tasks. Constructive logic combines the characteristics of logical formalisms with computational systems hence affording a bridge between the two: logical theorem proving techniques can be used to build up computational content.

This thesis discusses the following working systems implemented by the author:

- a program specialization system;
- a recursive program optimization system.

We also address the more general issue of program synthesis and program transformation through proof transformation. In particular, we discuss how, given computationally rich specifications, the *object-level* synthesis can be regarded as a transformation process, albeit interactive, which employs refinement rules to perform transformations.

We examine the relationships between proof constructs and the induced program constructs. Of particular importance to the optimization, and specialization, of recursive algorithms is the duality between mathematical induction and recursion.

In summary we may say that what, we believe, the proof synthesis environment offers us, over and above the more standard program transformation approaches to optimization, is the necessary tools to *represent* the procedural commitments and procedures within the proof that realizes the program being synthesized. By abstracting the main control features from the proof - superfluous to the functionality of the program - we are afforded a comprehensive *design record* of the program. By transforming those proof constructs associated with the efficiency, and not merely with correctness as regards the specification, we can automatically optimize the program. If the specification adequately states the defining input-output conditions of the desired program, then the optimization process is correctness guaranteed. Otherwise, the transformations are heuristically guided.

Transformation is achieved through the application of *proof transformation tactics.* Transformation tactics may apply mapping or transformation rules: *Mapping* rules are responsible for *recognizing* procedurally useful information and transformation rules modify the resulting mapped structures to achieve the desired target behaviour.

Techniques from the field of program transformation may be used to transform the computational content of a proof. An important technique for transforming exponential behaviour into linear behaviour is *tupling.* The MOPTS, unlike other existing transformation systems, performs this technique on (synthesis) proofs.

The novelty of the work described in this thesis stems from the following combination of properties exhibited by the author's transformation system:[13]

- Program transformation is approached through the transformation of *synthesis* proofs.
- These synthesis proofs are within a *constructive type theory.*
- The transformation system is implemented within the *same* constructive type theory.
- The system satisfies the following desirable properties for a transformation system:
  - *Correctness*: All transformed programs are correct with respect to their specifications.
  - *Generality*: The proof transformation system is highly flexible such that extensions to the existing system will allow for an open-ended number of function classes which can be successfully transformed.
  - *Automatability*: The source to target transformation requires no user guidance.
  - *Search and Control Features:* the system contains various properties, some of which are inherited from the object-level OYSTER system, that help to both guide search through the transformation space and to reduce that search space.
- The functionality of a program can be *automatically* adapted by the *correctness preserving* specialization (partial evaluation followed by pruning) of the corresponding synthesis proof.

We can summarize the main messages of this chapter as follows:

---

[13]There are also novel features specific to particular kinds of transformation. These will be discussed throughout the course of this thesis.

The OYSTER proof development system offers a flexible medium for program transformation since the corresponding synthesis proofs explicitly represent the procedural decisions, and choice commitments, associated with the programs.

An important commitment regarding the recursive behaviour of an extract program is the choice of induction schemata (and how the cases are satisfied).

By exploiting the common structure of OYSTER inductive synthesis proofs we can transform the induction schema employed in a proof yielding an inefficient program into a schema such that the new target proof yields a more efficient program. The process is fully automatic and meaning preserving.

1.6 Thesis Contents

We outline below the main contents of this thesis:

- In *Chapter 2* we provide a detailed description of the properties of the OYSTER system (tailored somewhat to the requirements of this thesis).

  We concentrate on the most important proof-program construct relationship as far as optimizing *recursive* algorithms is concerned: the duality between mathematical induction employed in synthesis proofs and the recursive structures which are induced in the resulting program. We discuss the common shape, or structure, of the majority of OYSTER inductive proofs which accounts for the generality of the inductive proof transformations.

  We provide a *user-directed* account of interactive theorem proving and, thereby, program synthesis using OYSTER. This is required for an understanding of the proofs and proof operations described throughout the thesis, and it also enables us to illustrate the various notational conventions used in the thesis. We ensure that the example syntheses employ mathematical induction such that the program construction rules associated with *recursion* are explained. Further more, the examples used are both inductive synthesis proofs of the

same program specification. This illustrates how the OYSTER system can be used as an *interactive* transformation system, comparable with some existing *non*-automatic program transformation systems. The example syntheses are chosen such that they accord with the automatic source to target example transformations described in *chapter 5.*

We discuss the notion of *correctness*, and what exactly is meant by *correctness preserving transformations.* This leads us to distinguish various meanings of *correctness* and to discuss the problem of *refinements capture.*

Finally, we outline the design and strategies of the MOPTS.

- *Chapter 3* consists of a review of current, and past, work on *program transformation* which has a relevant bearing to the techniques and strategies employed by the MOPTS. We first discuss some of the *general* methodological frameworks for program modification, such as analogy, explanation based learning, partial evaluation and the *fold/unfold* technique.

- *Chapter 4* is devoted to a discussion and analysis of the reconstructed, and extended, OYSTER *proof specialization system*, PSS, for adapting programs to special situations. Examples are presented and discussed. We compare the performance of the PSS with Goad's original specialization system.

- *Chapter 5* consists of a discussion and analysis of the OYSTER *proof transformation system*, OMTS. Examples are presented and discussed. We include a discussion of the CLAM *proof planner* – a system for automatically constructing *proof plans* which can then be used to guide the object-level proofs. This discussion concerns using the proof planner as a source of comparison – an *efficiency yardstick* – with the meta-level source to target proof transformations.

- *Chapter 6* contains a general conclusion and contains a summary discussion of the systems implemented for this thesis. We also give a summary of the advantages of the novel approach to program transformation (i.e. program

optimization via proof transformation, as opposed to program optimization via program transformation).

Finally, we provide suggestions for future research stemming from the discussions within this thesis.

Chapter 2

Program as Proofs, and Program Optimization Through Proof Transformation (An Overview)

2.1 Introduction

In this chapter we, first, describe the fundamentals of program synthesis, and verification, through proving theorems in the OYSTER system. We provide the requisite theoretical background concerning the following issues:

- Proofs as programs (the *Curry-Howard isomorphism*): program construction through constructive theorem proving.
- Martin-Löf type theory.
- Sequent calculus.
- Constructivism and types.
- Goal directed refinement:
  - introduction and Elimination rules;
  - mathematical induction, and the induction-recursion duality;
  - an example of *recursive* program synthesis through *inductive* theorem proving.

- Recursive efficiency: how different inductions realize different recursions (i.e. controlling recursive behaviour through induction).

- An example of how alternative *inductive* syntheses from common specification yields alternative programs that generate different *recursive* processes.

- Notational conventions for refinement applications and proof tree representation.

In the light of this discussion, we then go on to provide a (high-level) overview of the author's program transformation system, OMTS, which operates through the transformation of synthesis proofs:

- We first, by way of continuity, summarize all the properties discussed above, concerning program synthesis through constructive proof refinement, that have a direct bearing on the design and performance – correctness, automatability, and generality – of the OMTS concerning program optimization through constructive proof transformation.

- As a precursor to the subsequent two chapters, we describe the fundamentals of the system in order to provide the reader with a clear mental model of program transformation through proof transformation.

2.1.1 Rationale Behind Chapter 2

This chapter serves to provide the directly related theoretical background to the OMTS system descriptions, and performance discussions, provided in subsequent chapters.

Having provided a fairly detailed account of the relation between the various proof and program constructs, we can then present our notational conventions concerning OYSTER proof refinement, which will allow for a clear and simple reading of subsequent chapters without fear of omission regarding the underpinning synthesis of the ideas of constructive logic, Martin Löf type theory, refinement logic, and

logic programming. Thus, in subsequent chapters we can concentrate primarily on the effects of the individual OMTS transformations, without blurring the issues with the theoretical properties of "proofs as programs" that enable those effects.

The reason for including an overview of the OMTS system design immediately following the account of the OYSTER system is that, apart from affording the reader with a clear mental model from the outset, it also allows us to couch many of the properties of the OMTS in terms of those of the object-level OYSTER system, and ultimately in terms of the properties of constructive proofs based on Martin-Löf type theory.

2.2 The Duality Between Programs and Proofs

Constructive logic allows us to correlate computation with logical inference. This is because proofs of propositions in constructive logic require us to construct objects, such as functions and sets, in a similar way that programs require that actual objects are constructed in the course of computing a procedure.[1] Historically, this duality is accounted for by the *Curry-Howard isomorphism* which draws a duality between the inference rules and the functional terms of the λ-calculus (Curry & Feys, 1958; Howard, 1980). Since the terms, λ-*terms*, of the λ-calculus can be correlated with executable code then the duality is between inference rules and programs.

Such considerations allow us to correlate each proof of a proposition with a specific λ-term, λ-terms with programs, and the proposition with a specification of the program. Hence different constructive proofs of the same proposition correspond to different ways of computing a specific program specification. The reasoning for this can be set out as follows:

---

[1]Thus we can not, for example, compute (or constructively prove) that there are an infinity of prime numbers by assuming the converse and deriving a contradiction, rather we must produce a program that computes them (or a proof that we can always construct another one greater than the ones known so far).

1. Proofs of propositions correspond to terms of the appropriate *type*, such that,

2. the propositions are identified with the type of *their* proofs.

3. Proofs are closely correlated with the terms of the λ-calculus.

4. So by 2 and 3: propositions are identified with the type of the λ-terms, and

5. λ-terms can be equated with functional programs.

6. Therefore, by 4 and 5, the propositions can be viewed as *types* of programs.

7. In other words, the propositions of the λ-calculus can be correlated with descriptions (specifications) of programs which specify *what* task is computed by the program, and

8. the proofs of the propositions can be correlated with programs which determine *how* the task is computed.

9. Hence, numerous proofs of the same proposition can be correlated with numerous programs for computing the task specified by that proposition.

2.2.1 The OYSTER System

OYSTER is an implementation of a constructive type theory, based on Martin-Löf type theory, (Martin-Löf, 1979; Martin-Löf, 1984), and serves as a sequent calculus proof refinement system. OYSTER is written in Edinburgh Prolog, and run at the Prolog prompt level, so it is controlled by using Prolog predicates as commands. Proof tactics can be built as Prolog programs, incorporating OYSTER commands (which are simply Prolog predicates).

Prolog is used as the OYSTER meta-language for defining tactics. This is chiefly because the OYSTER proof mechanisms (and the CLAM proof planning mechanisms) can exploit the unification and back-tracking properties of Prolog.

The main benefit of using Martin-Löf type theory is that, recalling the previous section, it nicely combines typing properties with the properties of constructivism, such that we can both correlate the propositions of the λ-calculus with specifications of programs and correlate the proofs of the propositions with how the specification is computed.

The main benefit of using a sequent calculus notation, as opposed to that of any of the numerous natural deduction systems, is that at any stage (node) during a proof development, all the dependencies (assumptions and hypotheses) required to complete that proof stage are explicitly presented within a *hypothesis list.* A sequent is of the form [HYPOTHESES] $\vdash$ [CONCLUSION], where, in the course of proving the conclusion, refinements may either act upon the hypotheses, *elim* refinements, or act upon the conclusion, *intro* refinements. More detail concerning the nature of OYSTER refinement is covered in subsequent sections.

A major motivation behind the development of the OYSTER system is that the language uniformity of the logic programming environment allows for the construction of *meta-theorems* which express more general principles, concerning the object level theorem proving. This allows for the construction of programs, in Prolog, that manipulate proofs inside the system itself.

One such function is the construction of *tactics* which combine the object-level rules of the system in various ways and apply them to proof (sub)goals.

Another function, addressed in some depth in *Chapter 6*, is the CLAM automatic *proof planner* system which automatically constructs meta-level *proof plan* representations from proof specifications (Bundy *et al*, 1991). These proof plans can then be used to guide the object level synthesis/verification, with the advantage that the planning search space is considerably smaller than the object-level OYSTER search space.

A further function constitutes the backbone of this thesis: the construction of *meta-level* transformation tactics that operate upon the *object level* source (sub)proofs to produce target (sub)proofs from which optimized programs can be extracted. The fact that both source and target proofs yield programs that

behave according to the same specification is accounted for by the considerations of the previous section, specifically point 9.

The Nature of OYSTER Synthesis through Proof Refinement

Proofs are edited using a *refinement editor*, so-named because the proofs that OYSTER mechanises are *refinement proofs*. The OYSTER proof starts with the expression to be proved at the root of its proof tree, and constructs the tree back towards the leaves: the inference rules of the logic – *refinement rules* – are applied in reverse to a goal, to reduce, or *refine*, it to a set of sub-goals which, in turn, require proving in order to complete the overall proof. Thus, for example, if the user tells OYSTER to apply $\forall -$ *introduction* to a top-level goal statement, the system applies the rule in *reverse* – the effect of this is not to introduce, but to *remove* the the topmost connective (since the proof tree is being developed backwards).

Any proof is *complete* when the proof tree has been sufficiently developed *backwards* such that all leaves are accounted for – ie. when every leaf node can be proved without producing any further sub-goals. We refer to such proofs as being *goal-directed.* The refinement editor allows proof trees to be *traversed*, and refinement rules (or combinations thereof: tactics) to be applied to chosen nodes.

The end-nodes, or leaves, of a proof will always correspond either to axiomatic equalities, well-formedness goals or unification (i.e. where all components of the goal conclusion match with any of the proof hypotheses).

In practice, proofs are *not* created as single monolithic objects, but rather constructed from *definitions* and *sub-proofs.* As a result, OYSTER supports the construction not merely of single proofs, but of libraries of named proof objects – definitions (called *def* objects) and proofs (called *thm* objects). These can be loaded from, and saved to, ordinary UNIX files as desired.

The Extraction of Programs From Proofs

At any stage during the development of a proof it is possible to access the *extract term* of the proof constructed so far. I.e., each construct in the extract term

corresponds to a proof construct. As such, the extract term reflects the algorithmic ideas behind the proof of the theorem.

The extract programs consist of λ-calculus function terms, $\lambda(x, f_x)$ where f is the computed function and f_x the output when f is applied to input x. Since all type checking (well-foundedness checking) is done during the proof development then the extract terms need not, and do not, contain any typing information. The programs are hence fairly domain free and require no type checking during run-time. Those open subgoals in the proof which have sufficient constructive significance – i.e., not limited to well formedness goals – correspond to Prolog variables in the extract term.

The extract term of a theorem t is written in the form $term\_of(t)$. There is a built-in evaluator for type theoretic terms, which allows for the direct execution of OYSTER programs.

The existence of an extract term, corresponding to a particular proposition, is evidence that the proposition's type is inhabited, and this is equivalent to the proposition being constructively proved. All constructs of a completed proof that have an associated extract term of computational significance are collectively referred to as the *synthesis component* of the proof.

However, establishing that all the extract terms assembled from the synthesis component of a proof will indeed constitute a program that computes the specification embodied in the root node of a proof requires *verification*: the *verification component* of a proof is not used in executing the extract term, but ensures that the extract term satisfies the specification. Hence, although all components of a proof's extract term correspond directly to components in the proof, the relation is not bi-directional.

Ideally, as with conventional computational descriptions, the λ-calculus extract terms should only contain information about the function to be computed, where as proofs contain, in addition, information which is not concerned with simple execution. In practice, it is not so easy to (automatically) abstract away all the

verification information from the extract, although it is fairly clear when and where such information appears, if at all, within a complete proof extract term.

Constructivism and Types in OYSTER

Within type theory, each mathematical sentence is considered as a type, the elements of which are proofs of that sentence. A *type*, by definition, is a term which can be *inhabited* by other terms, or, equivalently, all types can have members.[2]

Types have fixed representations, equality relations and *type constructors* which allow for the construction of more complex types. The type constructors play analogous roles to logical operators such as conjunction, disjunction, implication, quantification, etc. Most importantly, a mathematical sentence is assumed to be true *if and only if* there is a proof of that sentence, that is, that the type is inhabited.

This property clearly illustrates the constructivist nature of the OYSTER logic: proofs require evidence which may take the form of constructing further proofs, sub-proofs, or the evidence may be provided by axiomatic truths. The evidence for a complete proof derives from the evidence of all its parts. The refinement rules are such that as a proof of a proposition is developed so the evidence associated with each node is accumulated until, upon completion, the proposition is completely satisfied. That is, through completing a constructive proof, each fragment has been provided with evidence, or a *witness*.

In general, to provide evidence for, or *satisfy*, any proposition we must provide evidence of objects that behave in the fashion specified by the proposition, and we must establish that each object inhabits a type (and thus that the proposition inhabits a type).

---

[2]For example, where as any individual natural number is *not* a type, the term *nat*, denoting a natural number, is a type since it is inhabited by the natural numbers $0, s(0), s(s(0)), ...$ (see next section).

So, for example, an implication $A \rightarrow B$ is assumed to be true *iff* one can construct a proof of B from a proof of A. A conjunction, $A \wedge B$, is assumed to be true *iff* one can can construct a (sub)proof for each of A and B. An existential statement, $\exists x{:}type.\, \phi(x)$, is true *iff* one can constructively prove that at least one element x has the property ϕ.

The hierarchical structure of Types

Types within OYSTER inhabit *universes*. These universes are hierarchically structured such that for any two universes, $u(i)$ and $u(j)$, then $u(j)$ can contain types within, or constructed from, $u(i)$ as long as $j > i$. This ensures that universes cannot contain themselves as elements, thus avoiding paradoxes arising from self-reference (e.g., *Russell's paradox*).

The Curry-Howard isomorphism can be restated as the principle of "propositions as types". This principle allows one to make "judgements" about specific well-formed formulae within the type theoretical formal system. Within the OYSTER system "judgements" take the form of two "membership criteria", each of which are required of proofs: *firstly*, for any object employed in a proof we must prove that it is *of* a certain type, and, *secondly*, all types must inhabit a universe, i.e., for any type employed in a proof, we must prove that it inhabits a certain universe (including the universes themselves).

The end-nodes, or *leaves*, of a proof tree will, generally, correspond to such "membership criteria". These usually cause the extract term *axiom* to appear within the completed extract program.

The first (smallest) universe in the hierarchy is $u(1)$. This contains all the primitive types:

- *Atom*: as in most conventional programming languages.

- *Nat*: a type for the natural numbers (with Peano Arithmetic).

- *Void*: this *empty* type provides the means to define contradictions and negation.[3]

- *Int*: provides the type for integers (with full integer arithmetic).

So just as "judgements" can be made concerning the type membership of an object, so "judgements" can be made concerning the universe membership of a type (e.g. $nat \in u(1)$).

The first universe, $u(1)$, also contains all possible types that may be constructed from the primitive types. For example:

List types: an object is of *list type* if it is any finite list over a single basic type.

Function types: these describe mappings from one type into another with the logical interpretation of implication.

Recursive types: these allow the construction of the recursive closure of a type scheme.

An important feature of the nat, int and list types is that they include the facilities for defining inductive terms over the respective types (details of which are given in §2.2.3).

To each type there corresponds canonical elements. The natural numbers, for example, have two canonical constants: 0 and the successor function s, such that if n is a natural number then $s(n)$ is canonical. In general, any element is canonical if its dominant function is canonical. Canonical elements cannot be reduced any further by evaluation, where as non-canonical elements, such as $\lambda x.s(x)$, can.

Regarding the equality of canonical elements, $s(i) = s(j)$ *in nat* if $i = j$ *in nat*, and $0 = 0$ *in nat*. OYSTER is able to evaluate equalities between non-canonical terms by appealing to the rules for the basic types.

---

[3]So $x : nat- >$ **void** is stating that x is not of type *nat*, and $x : nat, x : nat- >$ **void** is interpretated as a contradiction.

Evaluation

An extract term is executed using lazy evaluation: arguments within the extract term are only instantiated, and sub-terms evaluated, if they are required. Otherwise, the appearance of a "$\sim$" symbol within the extract term signifies an uninstantiated variable – i.e., an argument which is superfluous to the extracts evaluation.

The usual function of the *eval* predicate is to allow for either the direct execution of an extract term E, or for user supplied terms. The latter will usually take the form of $eval(E\ of Input, Output)$, where $Input$ is the user specified input and $Output$ the results of running E on $Input$. The former usage of *eval* is as a *program partial evaluator*: it is often useful to apply *eval* at least once on any extract term, thus $eval(E, Output)$, since it performs all possible evaluation of the extract *before* it is run on an instantiated input.

2.2.2 The Main Categories of OYSTER Refinements

We now describe the main kinds of refinement that are applicable within OYSTER paying particular attention to the kind of extract program construct that is witnessed by their respective application (thus building a picture of program synthesis through proof refinement).

Notation

In the *presentation* of the various rules, we shall use terms of the form $a : A$ where the label a denotes that there exists a proof, or evidence, for the type A. The rules will be presented upside down, with the goal sequent appearing at the bottom. This reflects the goal-directed nature of the sequent calculus.

In the *explanation* of the rules we shall sometimes "unpack" terms of the form $a : A$ into terms of the form P_{ϕ_A}. Such terms denote a proof, P, of A, from which the extraction term is ϕ_A, and where A, depending on context, may be a hypothesis or (part of) a goal conclusion (or equivalently, P_{ϕ_A} denotes that there

is evidence, ϕ_A, for A). Similarly, terms of the form $P_{\phi_A(a)}$ denote a proof of $A(a)$, with corresponding extract term $\phi_A(a)$ (or equivalently, $P_{\phi_A(a)}$ denotes that there is evidence for $A(a)$).

The term $A\ in\ Ui$ specifies that the type A inhabits some universe Ui at level i.

The *intro* refinements

The *intro* rules act upon (sub)goals, and have the effect of introducing their connectives in the conclusion formulae. Although the command *intro* can be associated with numerous ways of refining a (sub)goal, OYSTER is usually able to contextually determine exactly which operation is to be performed. In cases where ambiguities may arise some further syntactic sugar is necessary.

The inference rules of constructive logic need to reflect the structure of the proofs constructed by the rule application, where the resulting proof is considered as the extract term constructed during the proof process. We shall consider each of the main *intro* refinements in turn.

• Regarding logical conjunction, a proof of $A \wedge B$ is obtained by proving each of the conjuncts, A and B, separately and combining the resulting extract terms into an ordered pair. Hence the inference rule, $\wedge$-intro, is of the following form:[4]

$$\wedge\text{-intro:} \quad \frac{H \vdash A\ ext\ \phi_A \qquad H \vdash B\ ext\ \phi_B}{H \vdash A \wedge B\ ext\ \langle \phi_A, \phi_B \rangle}$$

where H represents the hypotheses, *ext* refers to the extension on the classical formulation of the rule (i.e., the *program extraction*), and ϕ_x is the extract term associated with a proof, P_{ϕ_x}, of X (or, equivalently, ϕ_x is the evidence that X inhabits a type). The OYSTER refinement rule for $\wedge$-intro is simply *intro*, and the

[4]The goal, at which the refinements are directed, appear below the horizontal line.

type of $A \wedge B$ proofs is the cartesian product $A\#B$.

• Similarly, the form of the inference rule for logical disjunction reflects that to prove $A \vee B$ we must prove either A or B:

$\vee$-intro:	$\dfrac{B\ in\ Ui \qquad H \vdash A\ \ ext\ \ \phi_A}{H \vdash A \vee B \quad ext\ \ inl(\phi_A)}$	$\dfrac{A\ in\ Ui \qquad H \vdash B\ \ ext\ \ \phi_B}{H \vdash A \vee B \quad ext\ \ inr(\phi_B)}$

The OYSTER refinement rule for $\vee$-intro is *intro(inl) or intro(inr)*, where *inl* and *inr* are indicators of which disjunct has a proof. The type of $A \vee B$ proofs is the disjoint union, $A \backslash B$, of the types of A and B.

• The form of the inference rule for logical implication reflects that to prove $A \supset B$ we must show how a proof, P_{ϕ_B}, of B can be constructed given a proof, P_{ϕ_A}, of A:

implication:	$\dfrac{A\ in\ Ui \qquad H, a{:}A \vdash B\ ext\ \ \phi_B}{H \vdash A \rightarrow B \quad ext\ \ \lambda a.\phi_B}$

where $\lambda a.\phi_B$ is a function such that the extract term for a proof of $A \supset B$ is a function of the hypothesis a. The OYSTER refinement rule for implication is *intro*, and the type of $A \supset B$ proofs is the type of all functions that output objects of type B given an input object inhabiting type A, i.e: $A \rightarrow B$.

Negation is a special case of $A \supset B$, where B is the empty type. The form of the inference rule for negation reflects that to prove $\neg A$ we must show that there is no proof construction, ϕ_{void}, for A, that is, A inhabits the empty type: $A \rightarrow void$,

negation:	$\dfrac{A\ in\ Ui \qquad H, a{:}A \vdash void\ ext\ \ \phi_{void}}{H \vdash \neg A \quad ext\ \ \lambda a.\phi_{void}}$

where A : *type* simply establishes that A inhabits some universe. The OYSTER refinement is again *intro*.

• The form of the inference rule for existential introduction, $\exists{-}intro$, reflects that to prove $(\exists x : A)B(x)$ one must provide an existential witness, a, for x such that there exists a proof, $P_{\phi_B}(a)$, of $B(a)$ with corresponding extract term $\phi_B(a)$. The proof construction is then the pair $\langle a, \phi_B(a) \rangle$.

$$\exists\text{–intro:} \quad \frac{H \vdash A \; ext \; a \qquad H \vdash B(a) \; ext \; \phi_B(a)}{H \vdash (\exists x{:}A)B(x) \quad ext \; \langle a, \phi_B(a) \rangle}$$

The type of such proofs is the disjoint union, $a : A \backslash B$, consisting of the types A and B, and labeled by the existential witness a. The corresponding OYSTER refinement is $intro(a)$.

• The form of the inference rule for universal introduction, $\forall{-}intro$, reflects that to prove $(\forall x : A)B(x)$ one must demonstrate that for any object, a, of type A, that there is a proof, $P_{\phi_B}(a)$, of $B(a)$.

$$\forall\text{–intro:} \quad \frac{A \; in \; Ui \qquad H, a{:}A \vdash B(a) \; ext \; \phi_B(a)}{H \vdash (\forall x{:}A)B(x) \quad ext \; \lambda a.\phi_B(a)}$$

Note that $\forall{-}intro$ is a special case of $A \supset B$. The OYSTER refinement is again $intro$.

The *elim* refinements

The *elim* refinements are the counterpart to the *intro* refinements, acting on hypotheses, rather than conclusions, so as to eliminate, rather than introduce, connectives. The *elim* refinements allow one to access (parts of) the hypotheses represented within a proof node hypothesis list. In general, the *elim* refinements are responsible for inducing in the extract algorithm one of the following: conditional decision functions, functions for accessing components of tuples, and substitution functions (of values for parameter names). We shall consider each of the main *elim* refinements in turn where, unless otherwise stated, the OYSTER refinement for elimination on a hypothesis h is simply $elim(h)$ (as with the *intro* refinements, OYSTER determines the correct usage from context).

• Regarding $\underline{\wedge - elimination}$, we must prove the following sequent $h : A \wedge B \vdash C$, where h labels the assumption of a proof of $A \wedge B$, corresponding to the program construct $\phi_{A \wedge B}$, and where C is the conclusion drawn from h. The OYSTER refinement is simply $elim(h)$ which enables us to access either of the program constructions, ϕ_A or ϕ_B, associated with the respective proof constructions of A and B. In other words, in the course of providing a proof, and there by extract ϕ_C, for C, $elim(h)$ enables us to provide evidence for $A \wedge B$ by appealing to the evidence for A and that for B (which necessarily exists in a constructive proof).

$$\wedge\text{-elim:} \quad \frac{A \; in \; Ui \qquad B \; in \; Ui \qquad H, a{:}A, b{:}B \vdash C \; ext \; \phi_C}{H, h{:}A \wedge B \vdash C \quad ext \; spread(h,[a,b,\phi_C])}$$

The proof construction is a *spread* function. The *spread* function takes a pair (first argument) and a list (second argument) specifying two labels and a term which may include them; on execution the function returns this term with the labels substituted by the elements of the pair.

• Regarding $\underline{\vee - elimination}$, we must prove the following sequent $h : A \vee B \vdash C$. This requires demonstrating that C can be proved with A as an assumption, yielding a program construction $\phi_{A \vdash C}$, and that C can be proved with B as an assumption, yielding a program construction $\phi_{B \vdash C}$.

$$\vee\text{-elim:} \quad \frac{A \; in \; Ui \quad B \; in \; Ui \quad H, a{:}A \vdash C \; ext \; \phi_{A \vdash C} \quad H, b{:}B \vdash C \; ext \; \phi_{B \vdash C}}{H, h{:}A \vee B \vdash C \quad ext \; decide(h, [a,\phi_{A \vdash C}], [b,\phi_{B \vdash C}])}$$

The extract term (or computational rule) corresponding to an application of $\vee - elimination$ is a *decide* term, where *decide* is a function which, upon execution, will yield its second or third argument depending on whether the proof of the first argument, $h : A \vee B$, appealed to a proof of the leftmost disjunct, via $intro(inl)$, or the rightmost disjunct, via $intro(inr)$.

• Regarding $\underline{\supset - elimination}$, we must prove the following sequent $h : A \supset B \vdash C$. This requires demonstrating that, given a proof of A, C can be proved assuming B.

$$\supset\text{-elim:} \quad \frac{B\ in\ Ui \qquad H \vdash A\ ext\ \phi_A \qquad H, b{:}B \vdash C\ ext\ \phi_C}{H, h{:}A \supset B \vdash C \quad ext\ \phi_C\{h(\phi_A)/b\}}$$

The hypothesis h names a function, corresponding to $A \supset B$, that produces a proof, P_{ϕ_B}, of B given a proof, P_{ϕ_A}, of A (i.e., $A \supset B$ produces a proof, b, of B given a proof, a, of A). Hence evidence for C is provided by substituting, within ϕ_C, all references to P_{ϕ_B} by the application of h to the extract term, ϕ_A, for P_{ϕ_A}. So the extract term corresponding to an application of $\supset -elimination$ is a substitution term (i.e. the substitution of $h(\phi_A)$ for b throughout ϕ_C).

• Regarding $\underline{\exists -elimination}$, we must prove the following sequent $h : (\exists x : A)B(x) \vdash C$. This will require access to the existential witness, a, for x, and access to the proof $P_{\phi_B}(a)$ of $B(a)$.

$$\exists\text{-elim:} \quad \frac{H \vdash A\ ext\ \phi_A \qquad H, a{:}A, b{:}B(a) \vdash C\ ext\ \phi_C}{H, h{:}(\exists x{:}A)B(x) \vdash C \quad ext\ spread(h,[a,b,\phi_C])}$$

The corresponding extract term is again a *spread* function which specifies the requisite accessories. Both a and $\phi_B(a)$ necessarily exist in a constructive proof.

• Regarding $\underline{\forall -elim}$, we must prove the following sequent $h : (\forall x : A)B(x) \vdash C$.

$$\forall\text{-elim:} \quad \frac{A\ in\ Ui \qquad B\ in\ Ui \qquad H \vdash A\ ext\ \phi_A \qquad H, a{:}A, b{:}B(a) \vdash C\ ext\ \phi_C}{H, h{:}(\forall x{:}A)B(x) \vdash C \quad ext\ \phi_C\{h(a)/b(a)\}}$$

$\forall -elim$ is similar to $\supset -elimination$: h is a function which yields a particular witness, a, of type A such that $B(a)$ holds. Since the purpose of $\forall$–$elim$ is to enable access to just such an instance of $B(x)$ then the extract term corresponding to its application is again a substitution term (i.e. the substitution of $h(a)$ for $P_{\phi_B}(a)$ throughout ϕ_C).

The OYSTER refinement is $elim(h,\ on(a))$

Case Analyses: the *decide* refinement

An important proof construct for synthesizing program decision procedures is the *decide* rule.[5] An application of $decide(P)$ sets up a case split at the proof node, where the two proof branches deal, respectively, with the cases P and $\neg P$, and where each split is responsible for synthesizing an extract sub-procedure ϕ_P or $\phi_{\neg P}$.

For example, if the case analyses corresponds to $x = y \vee x \neq y$, where x and y are naturals, then the refinement is

$$decide(x = y{:}nat,\ new[h]),$$

where h labels the proof hypothesis for the case split depending on whether or not $x = y$. The program construction associated with such a case analysis is of the form

$$nat\_eq(x, y, \phi_P, \phi_{\neg P}),$$

which specifies the required decision procedure: if $x = y$ then ϕ_P, otherwise $\phi_{\neg P}$. The hypothesis label h is instantiated either to $axiom$, if $x = y$, or to $\lambda\_.axiom$ if $x \neq y$, or more correctly if $x = y \rightarrow void$. The λ application on $\_.axiom$ provides a truth value for functions that range over the empty type $void$ (since if $x = y \rightarrow void$ then $x = y$ inhabits the empty type). The expression $\_.axiom$ simply signifies the absence of any objects. So the full program construct associated with the above application of *decide* can be represented thus (where we use $P_{\phi_{case}}$, rather than simply ϕ_{case}, to emphasize that a proof, P, of the witness ϕ at the particular case $case$ is required):

$$nat\_eq(x, y, P_{\phi_{x=y}}\{axiom/h\}, P_{\phi_{\neg x=y}}\{\lambda\_.axiom/h\}).$$

---

[5] Although closely related, the *decide* refinement used for decision procedures should not be equated with the *decide* extract term constructed through an application of $\vee$-elimination.

Entering New Facts into a Proof: the Cut (or *seq*) Rule

The generalized *seq*, rule allows one to enter, or *seq*uence, new facts into a proof by introducing a new node in the proof tree with two subnodes where:

- The first subgoal represents the original proof tree, P_{ϕ_C}, with the additional hypothesis that there is a proof, P_{ϕ_A}, of $h : A$. That is, a function, $\lambda P_{\phi_A}.\phi_C$, which given a proof of (evidence for) A yields a proof of (evidence for) C.

- The second subgoal is responsible for constructing a proof, P_{ϕ_A}, of the hypothesis h.

So the extract term associated with the application of *seq* is of the following form:

$$(\lambda P_{\phi_A}.\phi_C)(\phi_A).$$

The use of the *seq* rule is akin to synthesizing an auxiliary procedure for the program being constructed through the main proof.

We shall see, in *Chapters 5* and *6*, that the *seq* rule is ideally suited for optimizing a proof, that yields a program that computes a (relatively) inefficient procedure, by sequencing into that proof an efficient auxiliary procedure.

Lemmas

A further feature of OYSTER synthesis and theorem proving is the introduction of lemmas. Lemmas generally require their own proof and are used within the body of the main proof, usually to establish some property (computational or logical) required in the satisfaction of the main goal. The refinement associated with the use of a lemma depends on that usage. A lemma simply appears as a sub-theorem, of the form $term\_of(Lemma)$, within the extract term. Other than verification purposes, the author prefers to use the *seq* rule such that the lemma actually becomes part of the relevant theorem, thus ensuring that the global structure of the main theorem is kept as clean as possible.

2.2.3 The Induction-Recursion Duality

OYSTER provides primitive recursion schemas for the basic types: integers, natural numbers and lists. The recursion schemas enable one to:

(i) define recursive functions through case analyses, where the cases are determined by the structure of the type; and

(ii) apply induction as an inference (refinement) rule, thus enabling one to synthesize the dual recursion in the extract program.

We shall explain, through examples, both (i) and (ii) in turn. The explanation will cover three of the most commonly used induction schemas: *stepwise induction*, *course_of_values induction* and *list induction*.

Regarding (ii), we give examples of how two distinct inductive proofs of the same *complete* specification can yield two distinct recursive algorithms, each with identical functionality but differing considerably in the efficiency with which the output is *recursively constructed* from the input (§2.2.5 and §2.2.7). The main difference between the two proofs is that they employ, as inference rules, alternative induction schemes.

A preliminary in depth empirical study into the relative efficiency of a variety of *sorting* algorithms synthesized from the same specification, but using different induction rules, was conducted by the author prior to the research documented in this thesis (Madden, 1987b). This early research provided some initial comparisons between the efficiency of the recursion schemas associated with the various induction schemas employed during syntheses (if interested, the reader should consult the aforementioned reference).

Recursive Definitions

As an example of a recursive definition, we shall consider the *member* predicate which defines membership of a list of naturals. The *member* predicate is defined

over the naturals using *list induction*, *list_ind*, as the function constructor:

$$mem(e, l) \stackrel{def}{=} list\_ind(l, void, [hd, tl, rec, e = (hd{:}nat) \vee rec]).$$

Such a recursive definition is of the *list_ind* form: that is it allows one to refer to recursion over lists of natural numbers. The arguments in the body of the definition, which is simply an unpacking of the head, are identified below:

- The first argument, l, is the recursion argument.
- The second argument, *void*, is the (truth) value if the recursion argument is the empty list.
- The third argument, $[hd, tl, rec, e = (hd{:}nat) \vee rec]$ describes how to compute its value if it is of the form $hd :: tl$, where hd is the head of the list, tl the tail, and :: the infix list constructor. It is a quadruple of which the first three elements are hd, tl and the value, rec, of the function being defined when applied to tl. So, rec is, in effect, equivalent to the *induction hypothesis.* The fourth element, $e = (hd{:}nat) \vee rec$ is then the value of the function in terms of the other elements. In this case e is a member of l if $e = hd$ or if e is a member of the recursive tail tl (i.e. if $e = rec$ in the above definition of mem).

So, we can unpack the recursive definition thus:

$$mem(e, nil) \stackrel{comp}{=} false,$$

$$mem(e, hd :: tl) \stackrel{comp}{=} (e = hd{:}nat) \vee mem(e, tl),$$

where $\stackrel{comp}{=}$ means "*computes to ...*" or "*is computationaly equivalent to ...*"

Similarly, *p_ind*, specified thus:

$$p\_ind(x, y, [\sim, v, s(v)])$$

allows one to refer to (standard stepwise) recursion over the natural numbers, such that, for example addition, +, over the natural numbers is defined as

$$x + y \stackrel{def}{=} p\_ind(x, y, [\sim, rec, s(rec)]),$$

which states that if x is 0 then $x + y = y$, otherwise if $(x - 1) + y = rec$ then $x + y = s(rec)$, where s is the successor function.

Induction as a Refinement

Employing any of the induction schemas in a (synthesis) proof will induce the corresponding, or *dual*, recursion schema in the extract algorithm. So, for example, *stepwise* recursion over the natural numbers is synthesized by applying *stepwise* induction, conventionally represented thus (where s is the successor (constructor) function):

$$\frac{\vdash A(0) \quad ext \quad \phi_A(0) \qquad \forall y : nat.\, A(y) \vdash A(s(y)) \quad ext \quad \phi_{A(y) \vdash A(s(y))}}{\vdash \forall x : nat.\, A(x) \quad ext \quad \phi_A(x)}.$$

This states that A holds of any natural number, x, iff one can establish the following cases:

base case: A holds of 0,

step case: Assuming A holds of some natural number y, that P holds of $s(y)$.

The proof construction resulting from an application of *stepwise* induction is the *p_ind* construct shown in the previous section.

Stepwise induction on the naturals, along with *stepwise* induction on the integers and *list* induction, constitute the *primitive induction schemas*, and are built into the OYSTER system.[6] Employing such induction as an inference rule will split the proof into the corresponding cases. Each case will have a corresponding proof and extract component. The structure of the program extracted from the complete proof will mirror that of the (instantiated) dual induction schema. This is a general observation: to each induction schema there corresponds a dual recursion

[6]Although *list* induction can be interpreted as *stepwise* induction over lists (as opposed to the naturals).

schema. Hence a reliable heuristic that applies to synthesis through inductive theorem proving is that the behaviour of the induction variable should mirror that of the recursive terms in the function definitions.

More sophisticated induction schemas can be established by performing higher order proofs that appeal to the primitives in order to justify the scheme. An example of a non-primitive scheme is *course_of_values* induction.[7] As with the primitives, *course_of_values* recursion over the natural numbers is synthesized by applying *course_of_values* induction. This is conventionally represented thus:

$$\frac{\forall x : nat\ \forall y : nat.\ ((y < x) \rightarrow A(y)) \vdash A(x) \quad ext\ \ \phi_{((y<x)\rightarrow A(y))\vdash A(x)}}{\vdash \forall x : nat.\ A(x) \quad ext\ \ \phi_A(x)}.$$

Employing *course_of_values* induction as an inference rule does not automatically split the proof into a separate base and step case. Rather, the resulting subgoal represents the original proof tree with the induction hypothesis, $(y < x) \rightarrow A(y)$, entered into the proof as a new assumption (which tacitly includes the assumption that the hypothesis itself has a proof). The onus for splitting the proof into various cases, as defined by the function being synthesized, then lies with the user. This can be most elegantly done by employing the *decide* refinement to perform a case analyses (§2.2.2).[8]

We now illustrate the use of induction as a refinement. Since proofs employing *course_of_values* induction will form the source proofs for the (meta-level) proof transformations discussed in *chapter 6*, we shall choose this form of induction as our first example:

---

[7]Other non-primitive examples include *divide_and_conquer* induction and induction based on the construction of numbers as products of primes.

[8]The *seq* refinement could be used to enter a new fact which specifies the desired cases, but this is rather a roundabout solution.

Given the following (schematic) goal:

$$x : nat \vdash C,$$

then we can synthesize a program that computes C through *course_of_values* recursion by applying *course_of_values* induction over x. The *course_of_values* induction schema can be stated, in the λ-calculus notation, thus:

$$cv\_ind(x, [y, h, P_{\phi_C}(x)]).$$

The arguments of *cv_ind* are explained as follows:

1. x names the induction candidate: the argument over which the recursion is defined.

2. The second argument, $[y, h, P_{\phi_C}(x)]$, is a triple which defines the *recursive case* for the function being defined. The first two elements are y and h where:

   (a) y is any natural number *less than* the recursive argument (i.e. $y < x$). Hence, during the course of a proof, y can be instantiated to any desired value *less than* x. Furthermore, we can, depending on the recursive definition of the function being constructed, have multiple values for y (as long as each is less than x). This is, in effect, how cases can be introduced into a proof employing *course_of_values* induction.

   (b) h is the value of the function being defined when applied to y.

   (c) The third element of the triple, $P_{\phi_C}(x)$, provides the *step case* value for the function in terms of the first two elements, y and h, of the triple. Hence the third element, $P_{\phi_C}(x)$, computes the output value for the function/program being defined/synthesized.

Upon the application of *course_of_values* induction, the (sub)proofs corresponding to the base and step cases will supply the requisite proof objects: x, y and h (1, 2(a) and 2(b) above) will form hypothesis labels in the step case hypothesis list. These labels denote the existence of the induction variable, a natural number less than the x and the proof of induction hypothesis.

2.2.4 The Common Structure of Inductive (Synthesis) Proofs

Before we provide some examples, we shall say a little about the structure of the inductive proofs required to synthesize recursive programs.

Practically all inductive proofs follow the same *general* strategy as we have seen in the *course_of_values* example. This is one main reason why inductive proofs are a good candidate for transformation – due to the *similarity in form* of inductive proofs they afford us a *general* mechanisms for optimizing recursive algorithms.

So an important property of OYSTER inductive synthesis proofs is that they share a common structure or shape. This means that we can design a *typical* proof plan, **fig.** 2–1, wherein the key decisions and choice commitments made during a typical inductive proof are represented. These will involve applying one of the numerous induction rules and then witnessing the existential quantifier, using $\exists - intro$, at each of the induction cases (where, as indicated in **fig.** 2–1, the application of the *intro* rules are specific to inductive *synthesis* proofs). Finally, we must verify that the instantiated schema will yield a recursive schema that will compute the input-output relation specified in the main conjecture (represented in **fig.** 2–1 by $\forall x \ldots < exp >$). Note that we have indicated, within dashed boxes, that, following the witnessing steps of the (outermost) induction, we may wish to perform a further *nested* induction. These will take the same format as the outermost induction.

The verification stages will nearly always involve a process whereby formulae are "unpacked" - or *unfolded* - by replacing terms by suitably instantiated definitions.[9] The proliferation of this process such that recursive terms are grad-

---

[9]This unfolding is based on earlier work, notably (Aubin, 1975; Burstall & Darlington, 1977b). Our current version is a generalization of unfold which can use previously proved lemmas with a similar structure to unfold. Formal definitions of the (Burstall & Darlington, 1977b) usage are provided in *Chapter 3.*

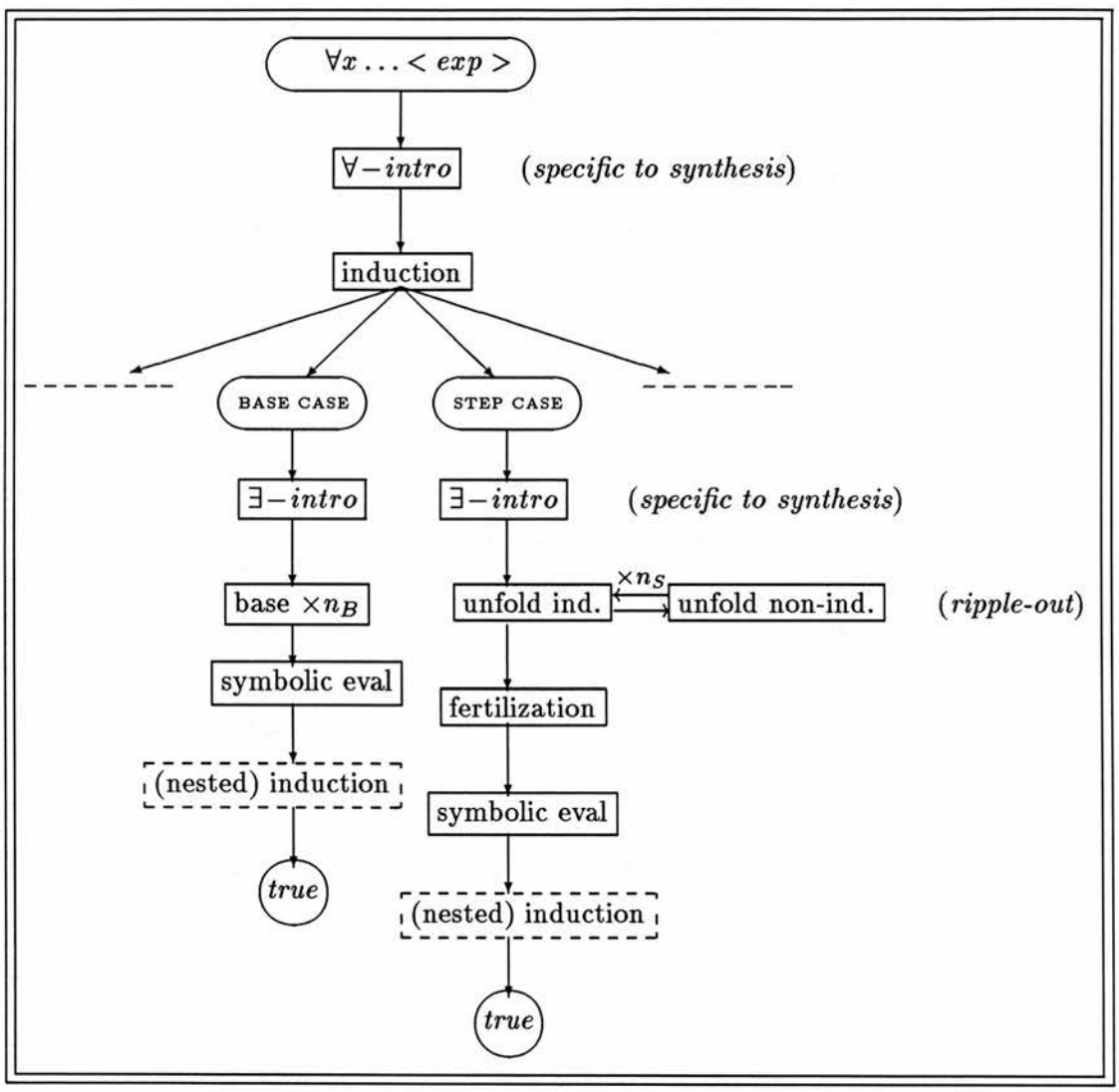


Figure 2–1: A general (typical) inductive proof plan (strategy)

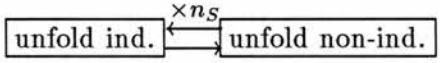
ually removed from the recursive branches – by the repeated unpacking of induction terms – is part of the (heuristic) process known as *rippling-out* (following (Aubin, 1975)). A simple examples of this would be the application of the following rewrites:

$$\begin{aligned} s(x)+y &\Rightarrow s(x+y), \\ append(e :: l_1, l_2) &\Rightarrow e :: append(l_1, l_2), \end{aligned}$$

where, in each case, the terms $x + y$ and $append(l_1, l_2)$ would unify with the respective induction hypotheses. Rippling-out may also involve the application of re-write rules that are not step cases of inductive definitions, for example:

$$\begin{aligned} even(x+y) &\Rightarrow even(x) \wedge even(y), \\ x \times y = y \times x &\Rightarrow x = y \wedge y = x. \end{aligned}$$

In **fig.** 2–1 we have schematically depicted the rippling-out process by the expression



where "unfold ind." refers to unfold re-writes that are step cases of inductive definitions, and "unfold non-ind." refers to unfold re-writes that are not step cases of inductive definitions.

The goal of rippling-out is to reduce the induction step case to terms which can be unified with those in the induction hypothesis. This unification process, performed at the induction step, has been coined *fertilization*. The rationale behind fertilization rests, again, on the induction-recursion duality. By ensuring that we include the induction hypothesis in the construction of the induction step, we ensure that recursion is built into the λ-function being synthesized.

The proof is completed when all the terms in the induction cases can be reduced to tautologies. Generally, this is done through the unification of terms in the sub-goals with those in any of the hypotheses made earlier in the proof, and by straightforward symbolic evaluation. Such reduction of the base case, involving

rewriting the equation at the base case with the appropriate definitional equations, has been represented in **fig.** 2–1 by the expression *base* $\times n_B$. The outermost arrows, directed at the broken lines, – – ––, signify that an induction schema need not be limited to a single base and step case.[10]

The key choices/decisions that differentiate, within the general strategy, one inductive proof from another are of the following nature:

(i) the choice of induction schema employed (e.g. *course_of_values* or *stepwise*);

(ii) the type of object introduced at the induction step (e.g. an object of type *nat* or an object of type *tuple*);

(iii) how the object is witnessed (instantiated) (i.e. the identity of the existential witnesses of the appropriate type chosen in (ii));

(iv) the subsequent verification of the instantiated schema (i.e. how the instantiated induction step is unfolded in order to facilitate fertilization, and the choice of any lemmas employed in the process).

Ramifications of the General Inductive Proof Strategy

The generality of inductive theorem proving has desirable ramifications concerning the expectations of the success of (meta-level) systems that manipulate such proofs. For example:

- The generality suggests the development of general inductive proof-planning mechanisms. This is precisely what is involved in the CLAM proof planner whereby successive rule applications are combined, by *tacticals*, into larger *tactics* and finally into a complete proof-plan – the actual combination desired being expressed in a formal meta-logic (*cf.* §6.3).

---

[10]The *course_of_values* induction, for example, has no built in limit on the number of cases, the onus resting on the synthesizer to construct the desired case splits (using the *decide* or *seq* rule).

- The generality also suggests the development of *general* inductive proof transformation mechanisms, the main topic of this thesis.[11]

In both cases, the system is designed to exploit these *general* properties of the (object-level) inductive proofs, hence increasing our expectation that if the system *design* works for a few examples then the same design will work for the majority.

2.2.5 An Example: Synthesizing an Exponential Process for Computing *Fibonacci*

To provide a clear illustration of the dual recursion induced in a proof extract by the application of *course_of_values* induction we shall discuss the fundamentals of synthesizing a program that computes the Fibonacci numbers.

Consider, from *Chapter 1*, the standard definition of the Fibonacci series:

$$\begin{array}{lll} f1: & fib(0) & \Leftarrow\!= 1; \\ f2: & fib(1) & \Leftarrow\!= 1' \\ f3: & fib(n+2) & \Leftarrow\!= fib(n+1) + fib(n). \end{array}$$

This can be reformulated as a specification, fib_{spec}, for computing the Fibonacci numbers:

$$fib_{spec}: \ \forall x \, \exists y{:}nat.\ fib(x) = y,$$

where fib is defined through the use of the following lemmas:[12]

---

[11]Indeed, we shall see that the proof representations abstracted from the OYSTER proofs by the OMTS bear a marked similarity to the CLAM proof plans (§2.3.3).

[12]The lemmas can be unproved without compromising the computational usefulness of the proof as a whole. However, this is not good practice since it *does* effect the status of the *synthesis* proof: if the synthesis component of the proof is to have a logical guarantee of meeting the specification, then any lemmas appealed to in the verification component must themselves be associated with a complete proof.

$$
\begin{aligned}
&fib\_lemma\ 1: \quad fib(0) \quad = s(0);\\
&fib\_lemma\ 2: \quad fib(s(0)) \ = s(0);\\
&fib\_lemma\ 3: \quad \forall x{:}nat\, \exists y{:}nat\, \exists z{:}nat\,.\, (x = 0 \rightarrow void) \wedge (x = s(0) \rightarrow void)\\
&\qquad\qquad \wedge fib(p(x)) = y \wedge fib(p(p(x))) = z \ \rightarrow \ fib(x) = y + z,
\end{aligned}
$$

and where p is the *predecessor* function defined by induction over the naturals, *p_ind*, thus:[13]

$$p(x) \stackrel{def}{=} p\_ind(x, 0, [a, \_, a]),$$

such that $fib(x-1) \equiv fib(p(x))$ and $fib(x-2) \equiv fib(p(p(x)))$.

The third lemma, *fib_lemma* 3, defines the recursive case and is naturally a course_of_values definition. Hence, since the behaviour of the induction variable should mirror that of the recursive terms in the function definitions then fib_{spec} is most naturally proved by *course_of_values* induction. However, alternative inductions can synthesize more efficient recursive behaviour. We shall discuss, and compare, such proofs in the subsequent section(s).

Basically the proof requires an initial application of the *intro* refinement, in order to introduce the universal quantifiers, followed by course of values induction on x.[14] The cases of the induction schema are then satisfied by setting up a nested case analysis structure using two applications of the *decide* rule, where the second application is nested within the first. The outermost case split corresponds to $x = 0 \vee x = 0 \rightarrow void$, and the innermost case to split to $x = s(0) \vee x = s(0) \rightarrow void$. By having the case splits nested in this way, we cover all the conditions specified in the course of values definition. By using the $\exists - elim$ rule, suitable witnesses are introduced at each case, and then verification is performed by appealing to (unfolding with) the relevant lemma (with various well-formedness goals being satisfied along the way).

---

[13] p is usefully employed as a destructor function of a function's data-structure (as opposed to using the canonical successor function, s to build constructor definitions).

[14] Recall that OYSTER proofs are *goal-directed*, thus rules such as $\forall - introduction$ have the *quantifier stripping* effect usually associated with $\forall - elimination$ in forwards proof systems.

We now give an account of the main steps during an inductive synthesis proof of fib_{spec}. We simplify somewhat both in the sense that we shall sometimes speak of a rule being applied which is, in fact, a tactic comprised of smaller rules and in the sense that we omit the well-formedness goals which are numerous and generally occur each time a new object is introduced into the proof. We augment the accounts of the main proof steps with diagrams which represent individual proof nodes. These will be numbered, N1, N2, N3,..., for reference purposes (i.e., regarding the actual proof, there may be further nodes between Nn and N$(n+1)$, such as the well-formedness goals).

Despite these simplifications, the overall proof structure is still fairly convoluted and so the reader can gain a clear overview of the complete proof structure, in particular the nested structure of the case analyses, by glancing now at **fig.** 2–3 §2.2.6.

• **Application of the *intro* Refinement:** $\forall -intro$

The initial *intro* refinement, on fib_{spec}, causes the existence of x to be added, as an assumption, to the hypothesis list. This assumption may then be referred to in the corresponding (and subsequent) conclusion(s). The resulting proof node will be of the following form:

N1

refinement :	$\boxed{intro}$
hypotheses :	$1.\, x{:}nat$
conclusion :	$\vdash_{\gamma} \exists y{:}nat\ fib(x) = y$
next refinement :	$\boxed{elim(x, cv)}$

The *hypothesis* slot contains the assumptions, or hypotheses, corresponding to the entries in the current (sub)goals hypothesis list.

The *conclusion* slot contains the current (sub)goal conclusion to be proved (with the aid of the aforementioned hypotheses). The Greek subscript is to be interpreted as the extract program associated with the particular branch of the

proof in which that goal appears. This will be the convention regarding all greek subscripts, such that separate proof branches have different subscripts according to the computation associated with that proof branch construction. In general, we depict the completed extract program by γ (hence, since the proofs are goal-directed, this will always appear at the root node of the proof rather than the leaves). The extract term(s) associated with the base case(s) of a proof will be signified by β, where we demarcate extract terms associated with different base cases by further subscripts thus: β_1, β_2. The step case of the proof will be signified by α.

For ease of reading, we have included two refinement slots: the *refinement* slot of a node N indicates the rule of inference whose application results in the hypotheses and/or conclusion at node N. The *next refinement* slot indicates the *next* rule of inference to be performed.[15] So the *next refinement* slot of a node Nn is equivalent to the *refinement* slot of node N$(n+1)$.

Since the (introduced) universally quantified variable, x, corresponds to the input over which the *fib* function will range, then x becomes a λ-calculus variable within γ. I.e., $\gamma = \lambda x.\_\_$, where $\_\_$ indicates the extract construction resulting from subsequent proof steps. The first of these, as indicated in the *next refinement* slot of the above proof node representation, is the application of the $elim(x, cv)$ refinement, which performs *course_of_values* induction on x.

• **Application of** $elim(x, cv)$**:** *course_of_values Induction*

Applying *course_of_values* induction on x will cause the induction variable and the induction hypothesis to be entered into the proof as new assumptions. The corresponding node is shown below.

---

[15]This is in keeping with the actual OYSTER interface.

N2

refinement :	$\boxed{elim(x, cv)}$
hypotheses :	1. x:nat
	2. H_{ind}:$\forall x'$:$nat.\ x' < x \rightarrow \exists y'$:$nat\ fib(x') = y'$
conclusion :	$\vdash_{\beta_1 \& \beta_2 \& \alpha} \exists y$:$nat\ fib(x) = y$
next refinement :	$\boxed{decide(x = 0\ in\ nat)}$

The *cv_ind* λ-calculus function will automatically build a recursion schema into the extract term being synthesized, with subsequent decisions made during the proof determining how the schema is evaluated. After the $elim(x, cv)$ application, the extract term, composed of the three constructs $\beta_1 \& \beta_2 \& \alpha$, will have been fleshed out to the following:

$$\lambda x.cv\_ind(x, [x', \phi_{H_{ind}}, \_]),$$

where x denotes the induction variable, x' denotes a natural number such that $\forall . x' < x$, and $\phi_{H_{ind}}$ denotes the constructive evidence for the induction hypothesis H_{ind}.

• **Application of** *decide*: *Splitting the Proof into Cases*

As indicated by N2, the next refinement is the *decide* rule. This first (or outermost) application of *decide* splits the proof into two sub-proofs, thereby introducing the two cases corresponding to whether $x' = 0\ in\ nat$ or $(x' = 0) \rightarrow void\ in\ nat$. Such a decision being prescribed by clause $f1$ of the *Fibonacci* definition (or by $fib\_lemma\ 1$).

The two new sub-nodes will be identical to the previous node (N2 above) except for the addition of either of the two aforementioned case conditions. The application of *decide* will introduce a *nat_eq* expression into the extract thus:

$$\lambda x.cv\_ind(x, [x', \phi_{H_{ind}}, nat\_eq(x', 0, \_, \_)]),$$

where the gaps, _, will be fleshed out by the case sub-proofs.

Regarding the node where x is assumed to be 0, an application of a simple rewriting tactic will substitute x for 0 in the conclusion. We omit this simple step in our proof node representations. So the case split resulting from the application of $decide(x = 0\ in\ nat)$ is represented by the following two nodes:

N3

ref. :	$\boxed{decide(x = 0\ in\ nat)}$
hyp. :	1. $x : nat$
	2. $H_{ind} : \forall x' : nat.\ x' < x \rightarrow \exists y' : nat.\ fib(x') = y'$
	3. $x' = 0\ in\ nat$
conc. :	$\vdash_{\beta_1} \exists y : nat\ fib(0) = y$
next	
ref. :	$\boxed{intro(s(0))}$

N4

ref. :	$\boxed{decide(x = 0\ in\ nat)}$
hyp. :	1. $x : nat$
	2. $H_{ind} : \forall x' : nat.\ x' < x \rightarrow \exists y' : nat.\ fib(x') = y'$
	3. $(x' = 0)\ in\ nat \rightarrow void$
conc. :	$\vdash_{\beta_2 \& \alpha} \exists y : nat\ fib(x) = y$
next	
ref. :	$\boxed{decide(x = s(0)\ in\ nat)}$

Here β_1, of N3, represents the extract computation for $fib(0)$, and $\beta_2 \& \alpha$, of N4, that for $fib(x)$ when $x > 0$.[16] We shall deal first with the proof N3 followed by that of N4.

- **Application of** *intro*: *Existentially Witnessing the First Base Case (*N3*)*

The existential witness, $s(0)$, for y' in the conclusion of N3 is provided by an *intro* rule application (as indicated by the *next refinement* slot of N3). Recall, from §2.2.2, that the proof construction for existential introduction is a pair consisting of the existential witness and a verification proof that the conclusion is true for that witness. The verification proof is provided by appealing to (unfolding with) *fib_lemma* 1, thus completing this branch of the proof. The □ symbol in N5 below signifies the completion (termination) of the proof branch (sub-computation) upon the application of the rule in the refinement slot.

---

[16] Where β_2 and α are the program constructs associated with the subsequent (second) base case and step case respectively.

N5

refinement : $\boxed{intro(s(0))}$

hypotheses : 1. $x{:}nat$
2. $H_{ind}{:}\forall x'{:}nat.\ x' < x \rightarrow \exists y'{:}nat\ fib(x') = y'$
3. $x' = 0\ in\ nat$

conclusion : $\vdash_{\beta_1} fib(0) = 0$

next refinement : $\boxed{lemma(fib\_lemma\ 1)}$ □

The extract term, with β_1 inserted, will now be:

$$\lambda x.cv\_ind(x, [x', \phi_{H_{ind}}, nat\_eq(x', 0, s(0), \_)]).$$

• 2^{nd} **Application of** *decide*: *Setting up the innermost case analyses*

The innermost application of *decide* on the conclusion of N4 splits the proof branch, corresponding to the outermost case condition $(x' = 0) \rightarrow void\ in\ nat$, into two further sub-proofs, thereby introducing the nested case split, corresponding to whether $x' = s(0)\ in\ nat$ or $(x' = s(0)) \rightarrow void\ in\ nat$. Such a decision being determined by clause $f2$ of the *Fibonacci* definition. This nested case split is represented by nodes N6 and N7 below.

N6

ref. : $\boxed{decide(x = s(0)\ in\ nat)}$

hyp. : 1. $x{:}nat$
2. $H_{ind}{:}\forall x'{:}nat.\ x' < x \rightarrow$
$\exists y'{:}nat.\ fib(x) = y'$
3. $x' = s(0)\ in\ nat$

conc. : $\vdash_{\beta_2} \exists y{:}nat\ fib(s(0)) = y$

next
ref. : $\boxed{intro(s(0))}$

N7

ref. : $\boxed{decide(x = s(0)\ in\ nat)}$

hyp. : 1. $x{:}nat$
2. $H_{ind}{:}\forall x'{:}nat.x' < x \rightarrow$
$\exists y'{:}nat.\ fib(x') = y'$
3. $(x' = s(0))\ in\ nat \rightarrow void$

conc. : $\vdash_{\alpha} \exists y{:}nat\ fib(x) = y$

next
ref. : $\boxed{elim(H_{ind}\ on\ p(x))}$

This application of *decide* will introduce a further *nat_eq* expression, corresponding to α, into the extract in place of the _ in the previous extraction ($\beta_2 \& \alpha$), thus:

$$\lambda x.cv\_ind(x, [x', \phi_{H_{ind}}, nat\_eq(x', 0, s(0), nat\_eq(x', s(0), \_, \_))]).$$

• **Application of** *intro*: *Existentially Witnessing the Second Base Case* (N6)

The sub-proof of N6 is completed in much the same way as that of N3: the existential witness for y' is introduced by applying $intro(s(0))$ (as indicated by the refinement slot of N6). The resulting conclusion, $\vdash_{\alpha} fib(s(0)) = s(0)$, is then verified by appealing to *lemma* 2. This yields an extract which now contains an output value for $fib(s(0))$:

$$\lambda x.cv\_ind(x, [x', \phi_{H_{ind}}, nat\_eq(x', 0, s(0), nat\_eq(x', s(0), s(0), \_))]).$$

• **Application of** *elim* **refinements:** *the step case*

The second branch of the innermost case-split, corresponding to the case condition $x' = s(0))$ *in* $nat \rightarrow void$, is the induction step of the proof, or, equivalently, the construction of the recursive call of the program being synthesized. This is represented by N7 above.

To digress briefly, the motivation behind the step case of the proof is to re-write the step conclusion into a form such that it is unifiable *either* with the induction hypothesis (H_{ind}), or with some subsequent derivation of H_{ind}. The former unification is dubbed *fertilization* and the latter *weak-fertilization.* Only by performing such fertilization goals can we assure that the λ-calculus extract being constructed is recursive. This is because the induction hypothesis is equivalent, within the λ-calculus, to the recursive call of the function specified in the root node of the proof. Hence by including the induction hypothesis in the justification of the step conclusion, we necessarily ensure that the function includes a recursive call during computation.

Returning to the development of the *Fibonacci* induction step, the task in the step case is to use the induction hypothesis, H_{ind}, in order to re-write the step conclusion of N7 into a form such that it can be verified by appealing directly to *fib_lemma* 3 (corresponding to the recursive case, $f3$, of the Fibonacci definition). This requires accessing the proof instances, or equivalently the program constructions, $\phi_{fib(p(x))}$ and $\phi_{fib(p(p(x)))}$ respectively, for $fib(p(x))$ and $fib(p(p(x)))$.

By adding the outputs of $\phi_{fib(p(x))}$ and $\phi_{fib(p(p(x)))}$ we then obtain an output for $fib(x)$.

The proof instances $P_{\phi_{fib(p(x))}}$ and $P_{\phi_{fib(p(p(x)))}}$ are obtained by performing two $\forall$: *elim* applications in succession on the induction hypothesis using the $elim(Hypothesis, on(Term))$ refinement, where $Term$ will take the value $p(x)$ and $p(p(x))$ respectively.

The proof construction resulting from each *elim* application will be a substitution term, §2.2.2, where $p(x)$ and $p(p(x))$ are substituted for the induction variable, x' in the induction hypothesis H_{ind}. Since both $p(x) < x$ *in nat* and $p(p(x)) < x$ *in nat* hold then, as H_{ind}, dictates we know there are proof constructions for each of the two evoked instances of H_{ind}: $P_{\phi_{H_{ind}}}(p(x))$ and $P_{\phi_{H_{ind}}}(p(p(x)))$ respectively.

These *elim* rule applications leave the extract term unchanged, although extra hypotheses are added, which may be used to construct a witness later on in the proof.

After the successive applications of $elim(H_{ind}\ on\ p(x))$ and $elim(H_{ind}\ on\ p(p(x)))$, we reach proof node N8.

N8

refinements : $\boxed{elim(H_{ind}\ on\ p(x))\ then\ elim(H_{ind}\ on\ p(p(x)))}$

hypotheses :
1. $x : nat$
2. $H_{ind} : \forall x' : nat.\ x' < x \rightarrow \exists y' : nat\ fib(x) = y'$
3. $(x' = 0)\ in\ nat \rightarrow void$
4. $(x' = s(0))\ in\ nat \rightarrow void$
5. $p(x) < x \rightarrow \exists y_1 : nat\ fib(p(x)) = y_1$
6. $p(p(x)) < x \rightarrow \exists y_2 : nat\ fib(p(p(x))) = y_2$

conclusion : $\vdash_\alpha \exists y : nat\ fib(x) = y$

next refinements : $\left(\boxed{2 \times supset-elim}\ \boxed{2 \times \exists-elim}\right)\ \boxed{\exists-intro(y_1 + y_2)}$

Since the proof is constructive, we must also establish the antecedents of 5 and 6, i.e., we must prove the facts that $p(x) < x$, and that $p(p(x)) < x$. In theory,

these would require first performing *supset− elim* on each of 5 and 6, in order to set up $p(x) < x$ and $p(p(x)) < x$ as separate subgoals, and then a couple of further (nested) inductive proofs to establish these sub-goals. In practice, however, such facts are proved and stored as lemmas which can then be invoked as required. We do not show a separate node representation corresponding to the establishing of such arithmetical truths in our node representations (although we do show the *supset− elim* applications as $2 \times$ *supset− elim* in the *next refinements* slot of N8).

Before any witness can be introduced for y, we first to eliminate the existential quantifiers binding y_1 and y_2. These two simple applications of $\exists$*–elim* are also not shown in a separate node representation (but are included in the *next refinements* slot of N8).

• **Application of** *intro*: *Existentially Witnessing the Step Case*

An output for the recursive case is now witnessed by evoking the induction hypothesis twice with inputs $p(x)$ and $p(p(x))$. We introduce an existential witness, $y_1 + y_2$, for y' comprised of the sum of the output, y_1, of hypothesis 5 and that, y_2, of hypothesis 6 (as indicated, again, by the refinement slot of N8).[17] By evoking the induction hypothesis twice in this way we induce *exponential* recursion into the extract. There is no way of avoiding this by using *course_of_values* induction to synthesize *fib_spec* (despite the fact that it provides the most natural way of synthesizing the program).

The current proof node, N9, is shown below.

---

[17]For clarity of presentation we use the infix notation, $i + j$, for addition, rather than the prefix functional form $plus(i, j)$. In practice, we use the latter by evoking the stored definition for *plus*, which is defined through *p_ind* induction over the naturals (*cf.* §2.2.3).

N9

refinement : $\boxed{\exists\text{-}intro(y_1 + y_2)}$

hypotheses :
1. $x : nat$
2. $H_{ind} : \forall x' : nat.\ x' < x \rightarrow \exists y' : nat\ fib(x') = y'$
3. $(x' = 0)\ in\ nat \rightarrow void$
4. $(x' = s(0))\ in\ nat \rightarrow void$
5. $y_1 : nat\ fib(p(x)) = y_1$
6. $y_2 : nat\ fib(p(p(x))) = y_2$

conclusion : $\vdash_\alpha fib(x) = y_1 + y_2$

next refinement : $\boxed{lemma(fib\_lemma\ 3)}$ □

The existential witness is then verified by first unfolding the conclusion with hypotheses 5 and 6, yielding the following conclusion:

$$\vdash_\alpha fib(x) = fib(p(x)) + fib(p(p(x))),$$

which is proved by appealing to *fib_lemma* 3, thus terminating the complete proof.

The λ-calculus proof construction will be $P_{\phi_{H_{ind}}}(p(x)) + P_{\phi_{H_{ind}}}(p(p(x)))$, i.e., the evaluation of that instance, hypothesis 5, of the induction hypothesis construction, where $x' = p(x)$, *plus* the evaluation of that instance, hypothesis 6, of the induction hypothesis construction, where $x' = p(p(x))$. So the corresponding extract construction, α, for the step case of the proof is $\phi_{H_{ind}}(p(x)) + \phi_{H_{ind}}(p(p(x)))$.

The Complete Extract Program for the *Course_of_Values* proof of *fib_spec*

The complete extract program, γ, results from the combination of all the separate proof branch constructions – which we denoted by the subscripts β_1, β_2 and α – appearing at the proof branch leaves of the first base case, second base case, and step case respectively. We indicate the input/output associated with each subscript (i.e., each case computation) in the λ-calculus representation below, **fig.** 2–2, of the complete extract program (where *nat_eq* has been abbreviated to *eq*, and *cv_ind* to *cv*). It is clear that there is a one-to-one correspondence between terms in the extract and terms in the proof from which it was extracted.

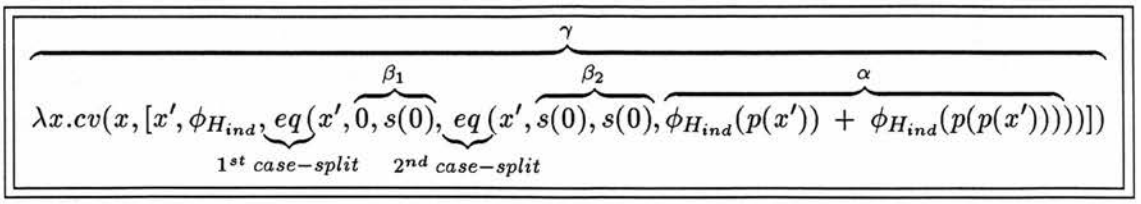


Figure 2–2: The *course_of_values* extract for fib_{spec}

However, it should also now be clear that the correspondence is not bi-directional: the *course_of_values* proof contained many steps which are not reflected in the extract program. Notably, due to the absence of anything resembling a hypothesis list, the extract program does not contain a record of the dependencies between facts involved the computation. Nor does it contain a complete representation of the verification component(s) of the proof. This provides a graphic illustration of how proofs contain information which is extraneous to that required for simple execution. In *Chapters 4* and *5* we describe how such information can be exploited, in different ways, to optimize the associated extract program.

Synthesizing *Tree Recursion* Through *Course_of_Values* Induction

The λ-calculus functional program extracted from the *course_of_values* inductive proof will compute the *Fibonacci* numbers according to the *course_of_values* definition ($f1, f2, f3$ §2.4.2 or, equivalently, *fib_lemmas* 1 to 3). The recursive pattern generated by this process was shown, for $fib(5)$, in **fig.** 1–3, *chapter 1.* Such a process is termed *tree recursive* since it resembles a tree where the branches split into two at each level. This is due to the fact that *fib* calls itself twice each time it is evoked.

Recall from *chapter 1*, that the *Fibonacci* tree recursive procedure generates an exponential process. Due to considerable redundant computation the process is inefficient: each recursive call is evaluated independently of the others thereby leading to repeated sub-computations. In **fig.** 1–4, *chapter 1*, we saw that, during

the computation of $fib(5)$, the entire computation of $fib(3)$ – almost half the work – is duplicated.[18]

The proof reflects the same inefficiency generated by the extract *course_of_values* recursive process. This could not be otherwise since the procedural commitments and/or decisions made during the synthesis determine the nature of the recursive process generated by the synthesized (extract) program. At the step case of the induction we appeal to *two* derivations, hypotheses 6 and 7 of N8, of the induction hypothesis, H_{ind}, hence indicating that the recursive process generated by the extract program will be exponential (or tree recursive).

Generally speaking, regarding the induction step of the *course_of_values* schema,

$$\forall x : nat,\ \forall y : nat.\ ((y < x) \rightarrow P(y)) \vdash P(x),$$

we will always obtain exponential complexity by invoking the hypothesis for a couple, or more, of values of y. However, if we invoke the particular value $y = x/2$, or indeed x/n *where* $n \geq 1$, then we obtain a logarithmic algorithm. So depending on what sort of recursive behaviour we desire in the extract algorithm, extra conditions are placed on how the *course_of_values* induction can be used in the proof.

We distinguish between *space* efficiency and *time* efficiency, although the above procedure generates an exponential process with respect to time, the *space* required grows only linearly with the input (since the process need only keep track of which nodes are above it in the tree at any point in the computation). In general, the time required by a tree-recursive process will be proportional to the number of nodes in the tree, where as the space required will be proportional to the maximum depth. Unless otherwise stated, we will for the most part be concerned with time efficiency.

---

[18]Indeed, it can be shown that, in computing $fib(n)$, the number of times the procedure will compute $fib(1)$ or $fib(0)$ is precisely $fib(n+1)$.

2.2.6 Convention for Proof Tree Representation

Throughout the thesis we shall schematically represent proofs so as to illustrate the structure – *branching pattern* – of the proof tree. As an example we represent, in **fig.** 2–3, the proof tree constructed during the the *course_of_values* synthesis proof of fib_{spec}. The nodes are almost identical to those presented throughout the explanation of the *course_of_values* proof. We have labelled the nodes corresponding to N1 and N2 merely to illustrate this fact. The only difference is that we use rather less cryptic names for the refinements than in practice.

Note also that in such representations of proofs we have, as a space saving device, omitted the initial $\forall$-intro application (since this will always occur as the first refinement of any synthesis proof). Hence the first sequent shown will consist of the $\forall$-introduced variable(s) (the input) as a hypothesis, or as hypotheses, and the relation between the input(s) and the (existentially quantified) output variable as conclusion.

2.2.7 An Alternative Means of Synthesizing *Fibonacci*: *Stepwise* Induction

We provide an alternative inductive synthesis of *Fibonacci* for three main reasons:

- To provide a further instance of the typical inductive proof, and one which employs a different induction schema to the previous example (thereby illustrating the key choices/decisions, (i)-(iv) §2.4, that differentiate one inductive proof from another).

- To further familiarize the reader with OYSTER proof refinement and notational conventions.

- **To clearly show how different recursive programs, with differing efficiency, can be synthesized from the same complete specification by employing different induction schemas.**

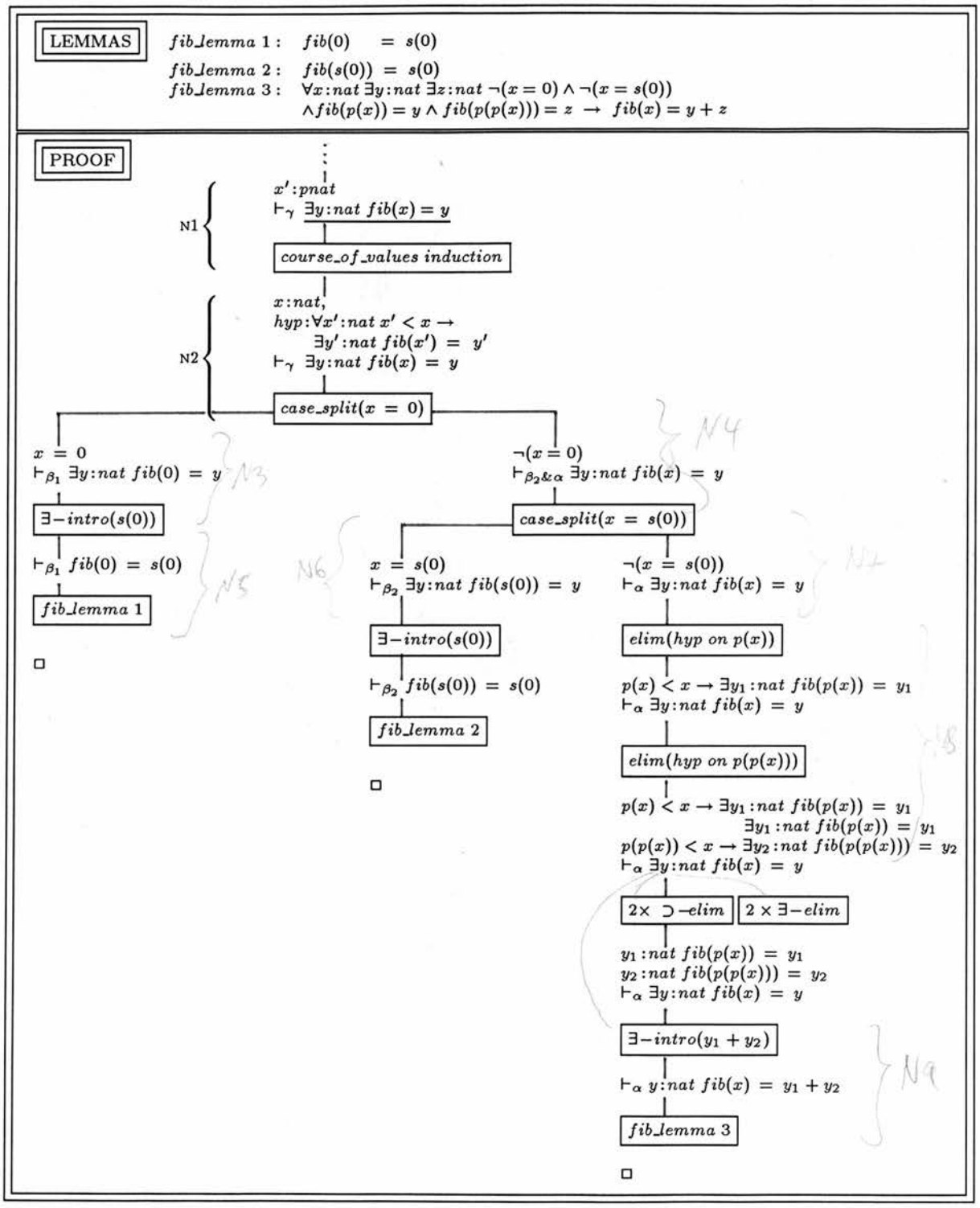


Figure. 2-3: Synthesis proof for *Fibonacci* using *course_of_values* induction

This sets the stage for *Chapter 5*, wherein we shall use the *course_of_values* and *stepwise* synthesis proofs of *Fibonacci* as our standard example source and target proofs of the OMTS transformations.

- Finally, to exhibit the inverse relation between the complexity of a proof and that of the program that it synthesizes.

We shall not, as with the previous example, give a "node by node" account of the synthesis proof. Rather, we provide a brief proof summary and then represent the synthesis by our schematic proof tree convention (as we did, in **fig.** 2–3, for the *course_of_values* proof).

One main advantage in using a *stepwise* scheme rather than using the *course_of_values* scheme directly, is that we can associate a particular complexity, namely *linear*, as upper bound to the algorithm (provided that the resulting recursion is the dominant influence on the overall complexity), whereas we have to constrain the *course_of_values* induction in order to do this.

The Use of Tuples

By employing *stepwise* induction over the naturals to synthesize a program that computes the *same* specification, fib_{spec}, as the previous *course_of_values* extract, will allow us to construct special tuples in order to evaluate the *Fibonacci* numbers. These tuples operate in such a way that potentially re-usable function calls – repeated computation – that appear in the tree recursive process generated by the *course_of_values* definition are grouped together, or *merged*, thus removing redundancy.

The *stepwise* induction is such that the first argument of the Fibonacci function is comprised of two further values (which are dealt with separately in the *course_of_values* induction). The induction works by constructing an auxiliary function, or *tuple*, $g(n)$ in terms of $g(n-1)$, where the first argument in both cases takes the "combined values" form.

The tuple takes a pair of arguments: the first corresponds to the sum of $fib(n-1)$ and $fib(n-2)$, i.e. $fib(n)$, the second corresponds to the first argument *of* the first argument, $fib(n-1)$. The tuple functional applys the addition function to the first and second arguments. So the goal $g(n)$ is ultimately satisfied by defining it in terms of the known course of values definition, i.e:

$$g(n) = \langle (fib(n-1) + fib(n-2)), fib(n-1) \rangle.$$

Note that the first tuple component is equivalent to the body of the *course_of_values* definition.

In effect, the tuple combines the values of the two step cases of the less efficient course of values definition. The result of tupling in this case is *linearization*: the production of a *stepwise* recursive algorithm which computes the Fibonacci function by a linear process.

The *Stepwise* Proof

In **fig.** 2–4 we display a schematic *stepwise* synthesis proof for *Fibonacci*, followed by a brief account of the main choices/decisions made during the proof. We simplify some of the rule applications in **fig.** 2–4 such that the proof operations are rendered more explicit. We indicate where such simplifications occur, and what has been omitted, in the brief account of the OYSTER synthesis that follows. This explanation, together with the previous discussion regarding **fig.** 2–3, will enable the reader to understand the subsequent proof tree representations used throughout the thesis.

Explanation of the *stepwise* Synthesis (with reference to fig. 2-4)

- The initial application of $\boxed{\forall - intro}$ causes the universally quantified variable x to be added as a new assumption. x will now appear in the extract term as a λ-variable, and is the input variable over which the λ-function will range:

$$\lambda x. \_$$

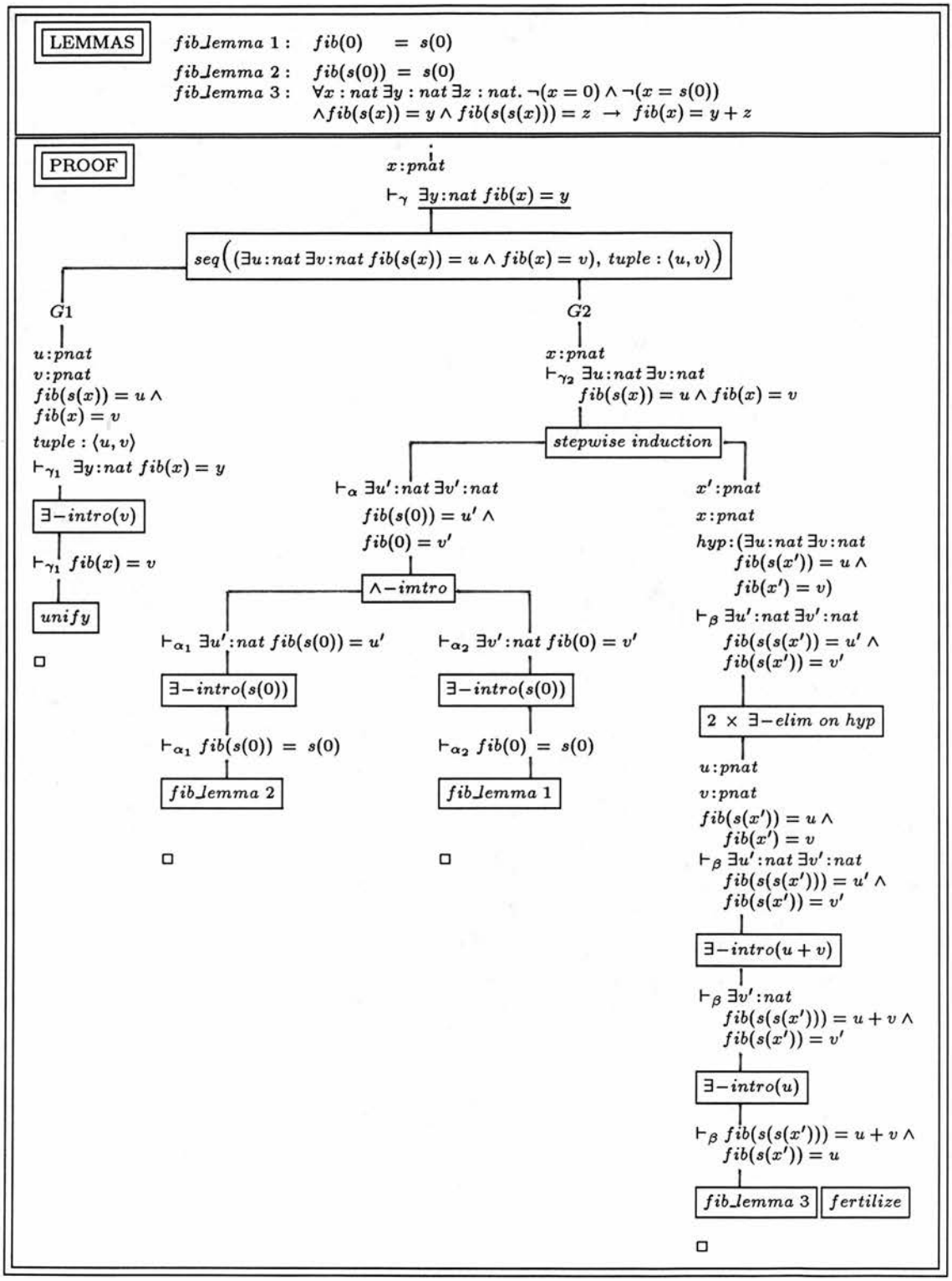


Figure. 2-4: Synthesis proof for *Fibonacci* using *stepwise* induction

- The $\boxed{seq\big((\exists u{:}nat\ \exists v{:}nat\ fib(s(x)) = u \wedge fib(x) = v),\ tuple : \langle u, v\rangle\big)}$ rule introduces a new node in the proof tree with two subnodes where one, $G1$, represents the original proof with an additional hypothesis and the other subnode, $G2$, is responsible for proving the hypothesis. Recalling §2.2.2, the proof construction corresponding to the application of the *seq*, or cut, rule will be $(\lambda P_{\phi_{G2}}.\phi_{G1})(\phi_{G2})$, where ϕ_{G1} and ϕ_{G2} represent the evidence for $G1$ and $G2$ respectively. The extract term constructed so far is

$$\lambda x.((\lambda tuple.\_)(\_)),$$

where the _'s will be fleshed out by the subsequent refinements. The extract thus far can be interpretated as a λ-function of x which returns a λ-function of a tuple, *tuple*. The first "_" signifies the construction that the function of *tuple* will return (*viz.* ϕ_{G1}), and the second "_" signifies the construction that the function of *tuple* is applied to (*viz.* ϕ_{G2}).
We now consider each sub-proof, $G1$ and $G2$, in turn:

(SUB)PROOF $G1$:

- The $\boxed{stepwise\ induction}$ rule applies *stepwise* induction, (§2.2.3), over the naturals, *p_ind* to x. As a result, the extract will now contain the (uninstantiated) *stepwise* schema:

$$\lambda x.((\lambda tuple.\_)(p\_ind(x, \_, [x', \phi_{H_{ind}}, \_])))$$

where x' is the induction variable, and $\phi_{H_{ind}}$ denotes the constructive evidence for the (stepwise) induction hypothesis H_{ind}. The first "_" within the *p_ind* term requires the subsequent construction of the base case output. The second "_" requires the subsequent construction of the step case output, which, for precisely the same reasons as for the *course_of_values* example, must necessarily involve H_{ind}, or some derivation(s) thereof, if the synthesized program is to be recursive.

- At the **base** case of the induction, $\boxed{\wedge\text{-}intro}$ separates the conclusion into the separate (base case) tuple components $fib(0)$ and $fib(s(0))$.

The subsequent two applications of $\boxed{\exists{-}intro(s(0))}$ witness a value for each component. These witnessing refinements on the base case cause the relevant _'s in the previous extraction to be instantiated thus:

$$\lambda x.((\lambda tuple.\_)(p\_ind(x, \langle s(0), s(0)\rangle, [x', \phi_{H_{ind}}, \_]))).$$

The witnesses for the two tuple components are then verified by appealing to lemmas *fib_lemma* 1 and *fib_lemma* 2 respectively.

- At the **step** case of the induction, the $\boxed{elim(H_{ind}, new[H_1, H_2, H_3])}$ rule strips both the existential quantifiers from the induction hypothesis and renames the bound variables such that $H_1 = u = fib(s(x))$, $H_2 = v = fib(x)$, and $H_3 = \langle H_1, H_2\rangle$. In constructive terms, this *elim* application yields a new derivation of H_{ind} which establishes that, given a proof

$$P\,\phi_{(fib(s(x'))=u)\wedge(fib(s(s(x)))=v)}$$

of the conjunction

$$(fib(s(x')) = u) \wedge (fib(s(s(x))) = v)$$

that there is constructive evidence, $\phi_{(fib(s(x'))=u)}$ and $\phi_{(fib(s(s(x)))=v)}$, for each of $fib(s(x')) = u$ and $fib(s(s(x))) = v$ respectively (i.e., that there is a proof for each conjunct of H_{ind}).

- The $\boxed{\exists{-}intro(u+v)}$ rule witnesses a value for u' of $u+v$. In practice, the rule application is $\exists{-}intro(H_1 + H_2)$ which, upon substitution of the hypothesis labels, H_1 and H_2, for their respective denotations, u and v, provides the requisite witness. This synthesizes a step case evaluation for the first tuple component $fib(s(s(x)))$.

  The existential witness is then verified by appealing to *fib_lemma* 3'. This lemma specifies a constructor definition for the recursive step of *Fibonacci*, and can be derived from the destructor version, *fib_lemma* 3, used in the previous *course_of_values* synthesis.

- $\boxed{\exists{-}intro(u)}$ is used to witness a value for the remaining, second, tuple component, $fib(s(x))$ which can then be directly verified by unification (fertil-

ization) with the induction hypothesis derivation $fib(s(x')) = u$. In practice, the refinement is $hyp(H_1)$ where *hyp* is a tactic which combines the $\exists$–introduction with the subsequent fertilization.
The extract will now contain a *spread* function (§2.2.2):

$$\lambda x.((\lambda tuple.\_)(p\_ind(x, \langle s(0), s(0)\rangle, [x', H_{ind}, spread(\phi_{H_{ind}}, [u', v', (u'+v')\wedge u']])))),$$

where the spread term specifies that the two components, u and v, of the pair (tuple), $\phi_{H_{ind}}$, whose existence is assumed through the induction hypothesis, are substituted, respectively, for u and v in the term $(u + v) \wedge u$.

(SUB)PROOF $G2$:

- An application of $elim(tuple, new[H_4, H_5, H_6])$ decomposes the tuple, synthesized via $G1$, into it's constituents: $H_4 = u = fib(s(x))$,$H_5 = v = fib(x)$, and $H_6 = \langle a3, a4\rangle$. For presentation purposes, we denote this simple *elim* step, in **fig.** 2–4, by a broken line thus – – – – – – –. This is primarily to prevent over-crowding of the diagram. The hypothesis list of the subsequent node clearly displays the derived hypotheses H_4, H_5 and H_6.

- Finally, $\boxed{\exists - intro(v)}$ is used to witness a value for y – the output for *Fibonacci* specified in the root goal. The resulting conclusion unifies directly with hypothesis H_5. In practice, the *hyp* tactic is used, thus $hyp(H_5)$.

The Complete Extract Program for the *Stepwise* proof of *fib_spec*

The *elim*, followed by the $\exists$–*intro*, application (on proof branch $G2$) will introduce a further *spread* function into the extract of the form $spread(\langle u, v\rangle, [\sim, y, y])$. This dictates that the output for *Fibonacci* is obtained by substituting the second element of the tuple, synthesized through $G1$, for y in the root node specification. This completes the program construction. Note that the *stepwise* extract, as in the *stepwise* proof, contains only a single evocation of the induction hypothesis, H_{ind}. The recursive process generated by the *stepwise* extract is hence linear.

It is the use of tupling which allows us to construct such a linear process: the solution for *Fibonacci* corresponds to v in the above extract (i.e., the second

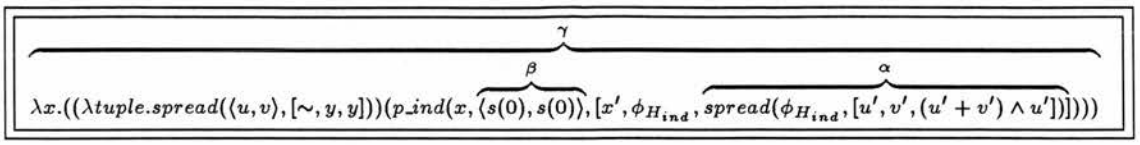


Figure 2–5: The *stepwise* extract for fib_{spec}

argument of the first tuple component). Parameter u acts as an accumulator since its value in successive invocations accumulates the value(s) of the function. So, the process generated is *linear recursive* since, with u and v initialized to 1 and 0 respectively, the procedure applies the simultaneous "transformations" shown on the l.h.s. of the following informal equivalence (where $A \longmapsto B$ means A "transforms" to B),

$$\left\{ \begin{array}{lcl} u & \longmapsto & u+v \\ v & \longmapsto & u \end{array} \right\} \equiv \{\langle u,v\rangle \longmapsto \langle u+v,u\rangle\} \textit{ where } u = fib(i) \textit{ and } v = fib(i-1), \textit{(for some } i\textit{)}.$$

This represents a single recursive call where to obtain $\langle u+v,u\rangle$ we require a single evocation of the induction hypothesis construction, corresponding to $\langle u,v\rangle$.

So after applying this "transformation" n times then u and v will be equal to $fib(s(n))$ and $fib(n)$ respectively, i.e., (schematically),

$$\left\{ \begin{array}{lcl} u & \longmapsto & u+v \\ v & \longmapsto & u \end{array} \right\} \times n \equiv \langle fib(s(n)) + fib(n), fib(s(n))\rangle.$$

2.2.8 Discussion

By synthesizing a *stepwise* program that satisfies the same complete specification, *fib_spec*, as the *course_of_values* program we have, in effect, performed an interactive optimization – or *linearization* – of the *course_of_values* program (albeit at the object-level of the proof refinement system). So by observing the structure of a proof which yields an inefficient program, we can refine a proof from the same complete specification which yields a more efficient algorithm.

In *chapter 5* we describe how the more efficient algorithm of each example pair can be obtained, without doing the synthesis from scratch: by exploiting dependency information represented within the proof node hypothesis lists we are able to *automatically* transform, at a meta-level, the proof associated with the less efficient algorithm.

The use of tuples to group together potentially re-usable function calls – the *tupling technique* – has previously been investigated, not within the proofs as programs context, but within the context of the *fold/unfold program transformation strategy* (Pettorossi, 1984; Burstall & Darlington, 1977a; Chin, 1990). We shall survey both the tupling and fold/unfold program transformation techniques in *Chapter 3.*

In *Chapter 6* we discuss the linear to logarithmic transformation of the *stepwise Fibonacci* proof. This illustrates how successive optimizations of a program can be achieved by successively transforming the corresponding inductive synthesis proofs in accordance with the complexity ordering of the various induction schemas.

Inverse Complexity Relation Between Proofs and Programs

As a rule of thumb, the complexity of the proofs associated with each of the induction schemata employed in the alternative syntheses of fib_{spec}, including the *divide_and_conquer* inductive proof, vary inversely with the complexity of the corresponding recursive process invoked in the associated extract program. This means that transformations which increase the syntactic complexity of the source *course_of_values* proof, by performing induction schema transformations, will decrease the complexity of the recursive behaviour of the extract programs (from exponential to linear).

Hence, as remarked in *Chapter 1*, proof transformations allow the human theorem prover to produce an elegant *source* proof, without clouding the design process with efficiency issues, and then to transform this into an opaque proof that yields an efficient *target* program.

This relation is something which merits further attention but for which, as of yet, there is only empirical justification and a quasi-theoretical foundation. We shall not, in the main body of this thesis, discuss at length any such theoretical foundation (although we do address it briefly in *Chapter 4*). Intuitively speaking, however, the extra complexity associated with a target proof can be thought of as additional information required to compute the specified input/output relation *efficiently* as opposed to simply ensuring that the specified input/output relation is computed.

2.2.9 Refinements Capture and Correctness (or *What's in a Specification?*)

Much of the work on correctness falls within the domain of mechanical theorem proving, since the main task is to show that the program agrees with the specification (to this end, correctness also involves finding methods for attaching parts of the specification to the code). This is also directly related to the problem of verifying a program, since a program is verified with respect to its specification. That is, the notion of program correctness presupposes a precise specification since *a program is only correct relative to its specification.*

But how do we know whether or not a specification totally and unambiguously captures the *desired* program input-output relation? And captures it in such a way that the output is computed efficiently from the input? This is the problem of *Refinements Capture*, so called because the problem concerns capturing, within the *specification content*, the right information such that subsequent refinements will construct the user-desired input/output behaviour (where efficiency of the computation is one desirable attribute). Where we categorize the *content* of a program specification in the following two senses:

1. *Complete* or *incomplete*: a specification is complete if it totally and unambiguously captures the *desired* program input-output relation. I.e., a complete specification should completely determine the function to be computed, otherwise the specification is incomplete.

2. *Full* or *weak* (*under specified*): a specification is full if it specifies everything we wish to specify. This may not include everything required to completely determine the function. Hence a specification may be full without being complete. For example, if *Fibonacci* is defined by

$$f(1) = 1,\ f(2) = 1,\ \forall x > 0,\ f(x) + f(x+1) = f(x+2),$$

where there is no value assignment for 0 then this may be a full specification as far as the user is concerned, but it is incomplete.

A specification is weak if it captures little about the program which satisfies it. For example, a program may be extracted from a proof for which the specification only captures the *constructive type* of the input and output.

So refinements capture has an important bearing on *both* the object-level synthesis *and* the meta-level OMTS transformations:

A. Synthesis: only if we are certain that the specification, S, is complete (i.e. totally and unambiguously captures the desired input-output relation), do we know that a program extracted from the proof of S will compute the desired input-output relation.

B. Transformation: only if we are certain that the specification, S, totally *and* unambiguously captures the input-output relation can we ensure that the *functionality* of both the source and target programs, extracted from source and target proofs satisfying S, are the same.[19]

There are also three usages of "correctness" that we wish to distinguish, depending on the context within which the term is applied:

---

[19]This is not a problem for *functionality transformation*, such as the specialization of programs through proof transformation, where the source specification is transformed along with the proof. The target program, however, should still be correct with respect to the target specification in the sense of **A**(*cf. Chapter 1, section 3.1, Chapter 3, section 3* and *chapter 5*).

1. Program Correctness: This corresponds to the usage of correctness in A above, *the correctness of an extract program relative to its synthesis proof specification.* The source and target programs of a transformation should be correct with respect to their respective specifications (which may differ in the case of transformations on a source programs functionality).

2. Source to Target Correctness: This corresponds to the usage of correctness in B above: *the correctness of a target program relative to the source proof specification.* By this meaning, an optimization process designed to preserve the functionality of the source program fails if the target proof *does not* satisfy the source specification, *regardless of whether or not it satisfies the target specification.*

3. Well-formed Correctness (or Meaning Preservation): Any individual (sub)-transformation on a source proof (sub)construct is correct in this sense if it only produces *legal* target proof constructs: the post-condition for well-formed correctness is that *after applying an individual transformation operator, the current status of the target proof is either partial or complete.*

Refinements capture is directly concerned with 1 above: program correctness. However, as implied by A and B above, it is only by ensuring specification correctness, with respect to a source synthesized program, that we can ensure the Source to Target Correctness of the optimization of the source.

The problem of refinements capture is one that needs to be addressed by the whole automatic programming community. Unfortunately, this has not been the case and there is little in the literature which addresses the problem head on (although there is plenty concerning the creation of new specification languages). In the near future the automatic programming community will *have* to address the problem of refinements capture since it would seem that much of the human effort removed by automatic programming is shunted on to the problem of totally and unambiguously specifying a procedure and in a way that will produce efficient implementations.

However, this is not a thesis specifically about refinements capture, and we shall overlook the refinements capture problem, assuming, as in the relevant literature, that it is fairly obvious when a specification is full or weak, complete or incomplete. The purpose of this brief section was simply to state explicitly, within a body of work concerning automatic programming, that there *is* a problem here, and one which requires attention sooner rather than latter.

We shall also, unless otherwise stated, remain with the standard usage of *correctness*: the correctness of an extract program *relative to its synthesis proof specification* (i.e. 1 above).

2.2.10 Reacting to Changing Specifications

The previous section leads us nicely on to a further ramification of using a *specification language* to synthesize, or transform, programs. Suppose we have some synthesized source program that we wish to adapt to operate efficiently on a specific value (range) of input(s), that is, we wish to modify what the specification captures. By modifying the program specification in accordance with some *desired* adaptation, we can then propagate the modification through the proof until we achieve a complete modified target proof that yields an adapted – or *specialized* – program. Within the confines of the target input range (value), the specialized program will then operate more efficiently than the general source. As a simple example, suppose the adaptation is of a program which computes some function f and is extracted from a proof of the following schematic specification, where n is some known value:

$$\forall x \exists y \exists z . x \leq n \rightarrow f(x) = y \vee x > n \rightarrow f(x) = z.$$

If we know that, within some application field, $x > n$ is always true then we could adapt the specification to

$$\forall x \exists z \rightarrow f(x) = z,$$

and modify the source proof by, essentially, removing the case split corresponding to $x \leq n \vee x > n$ and replacing it, implicitly or explicitly, by the (sub)proof associated with the case condition $x > n$.

Alternatively, if we can determine that one of the disjunct proof constructions is *redundant*, with respect to computing f, then we can perform a similar replacement of the case split.

Such considerations constitute the basis of the OMTS specialization transformations.

2.2.11 Exploiting the Properties of OYSTER Synthesis for Proof Transformations

Having covered the fundamentals of OYSTER synthesis, we now itemize the main properties of the OYSTER refinement system that are exploited by the OMTS. This serves both as a summary of the previous sections and as a precursor to the following sections outlining the actual OMTS design.

1. Since we can correlate propositions with program specifications, and proofs of propositions with functional (λ-calculus) programs then we can obtain various procedures for computing the same function, specified by a specific proposition, by performing numerous proofs. So by transforming a proof of a proposition into a different proof of the same proposition we can transform programs through proof transformation.

2. Proofs, unlike programs, contain additional information which is not concerned with simple execution. Notably, they contain an account of the dependencies between facts involved in the computation of the extract term, *and* information concerned with verifying that the extract term computes the task described by the specification. This information can be exploited in numerous ways for the automatic transformation of programs through proof transformation.

3. Since the extract algorithm is guaranteed to compute the input-output relation specified in the proof specification, then this affords us with a correctness guarantee for proof transformations.

4. We can correlate the recursive behaviour of an extract program with

   - the induction schema employed (eg. *course_of_values* or *stepwise*), and
   - the manner in which the induction hypothesis of the (instantiated) schema is evoked during the satisfaction of the induction conclusion.

   This provides us with a direct handle with which to modify the efficiency of the process generated by a recursive (extract) program. Considerations relating to 2 above allow such modifications to be done automatically through proof transformations. Considerations relating to 3 above allow such transformations to be performed with a correctness guarantee.

5. Unlike most program refinement systems (including the majority of recursive equation development systems) OYSTER has a termination condition: namely the termination of a proof when all (sub)goals are satisfied. Similarly, the program through proof transformations have a termination condition: namely the termination of a *target* proof when all (sub)goals are satisfied.

6. We can also transform the input/output conditions of a source program by transforming the source proof specification and then propagating the modification throughout the proof. Correctness of the target, with respect to the modified specification, is guaranteed (3 above). Usual program transformations do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour.

7. Expectations of success, together with the generality of design, are desirable features of any transformation system. We have seen that synthesizing recursive programs invariably requires mathematical induction, and that the

majority of inductive proofs share the same proof strategy, or structure. Hence, although the actual implementation of the OMTS should be regarded as embryonic, the success that it has achieved, together with the aforementioned properties of OYSTER proof synthesis, suggest that a far broader corpus of recursive programs, than those covered in this thesis, can be optimized using the same system *design* (described in outline in the following section, and in depth in subsequent chapters).

8. Furthermore, due largely to the *general* strategy of inductive theorem proving, the ability to provide typical proof strategies, e.g., for the rippling-out stages of verification, assists with the automatic completion of the target proofs.[20]

2.3 Mapping and Transforming Proofs (an Overview of the OMTS)

In the remainder of this section we provide an overview of the *design* of the OMTS. This serves as an introduction to specialization, *Chapter 4*, and optimization, *Chapter 5*, which both share the same central transformation mechanism.

2.3.1 Transformation Tactics

Recall that the OMTS exploits the properties of the object-level OYSTER proofs in the following ways:

1. The choice of proof constructs made during synthesis have associated with them the procedural commitments which can be abstracted from the proofs.

---

[20]Regarding this, and the previous item, it is also the common structure property of the proofs that accounts for the automatability, and anticipated future success regarding generality, of the CLAM proof-plan research also being conducted at the Edinburgh AI department (*cf. Chapter 6*).

2. *Dependency information* relating such procedural commitments can also be abstracted from the proofs.

This enables us to identify the OMTS transformation of OYSTER proofs by the following:

> Transformation = The application of transformation tactics to modify the procedural commitments made during a source proof synthesis *guided* by dependency information represented within the proof.

2.3.2 Categorizing the Modifications

For the purposes of explanation it is beneficial to categorize the transformation tactics into two sorts of "rules": *mapping rules* and *transformation rules.* In practice, however, it should be remembered that the same rule, or operator, may be used for either mapping or transforming.

Mapping: Mapping concerns transposing, or *abstracting*, a portion of the source problem to be used in developing the target with little or no alteration prior to testing the completed target.

Transforming: Transforming concerns actually altering, or adapting, a portion of the source problem before it is used in the developing target proof.

So, *mapping* is responsible for *recognizing* procedurally useful information, and *transformation* is responsible for *modifying* the resulting mapped structures to achieve the desired target behaviour.

The OMTS is tuned to recognize the key positions within inductive proofs that have a decisive effect on the recursive behaviour of the extract algorithm. These key positions correspond to the application of an induction rule, the constructive type of the objects required to witness the induction cases, the actual proof constructs introduced to witness the induction cases, and finally the definitions chosen to complete the verification component of the proof.

The OMTS does not actually modify the original source proof, although this could be easily done, but rather constructs a target proof from scratch by mapping across the source specification, and then using the source proof as a guide in developing the target proof (the actual transformation being performed through the modification of special OMTS proof abstractions – §2.3.3). This usually consists of the following:

1. Noting what induction rule is applied in the source.

2. Using the source proof equational definitions, together with specific transformation techniques, to introduce a target proof object of the required type (for example, linearization requires using the tupling technique to introduce a target *tuple* object of a specific size).

3. Applying a target induction with a more efficient computation rule (e.g., applying *stepwise* in place of *course_of_values* induction).

4. Abstracting all the information from the source proof which can be exploited in completing the target induction:

   (a) mapping across all the structures in the source proof which will prove useful for instantiating the target schema. These structures will consist of specific hypotheses, rule applications and sub-goals.

   (b) Abstracting dependency information from the proof (i.e., the interrelations between (sub)goals, assumptions and hypotheses).

5. Both 4(a) and 4(b) may then be used to guide the witnessing of the existential variables at the target induction cases.

6. Finally, 4(a) and 4(b), together with a knowledge of the general strategy required to verify inductive proofs, is used to guide the verification of the instantiated target induction cases. The general strategy will invariably consist of unfolding the sub-goal at the target induction step with the source proof equational definitions, until a match (fertilization) is found with the target proof hypotheses.

Although the transformations involve using the source proof to *guide* the new construction of a target proof by mapping, and then transforming, portions of the former, the source proof, and extract, is itself preserved. This is an intentional design factor since, for some applications, it may prove desirable to have access to both the source and target proofs at the termination point of the transformation. An example of this is the specialization transformations of the OMTS, wherein the user may well wish to retain the source proof, for general applications, as well as obtain a specialized proof, for specialized applications. So for practical purposes, it may prove useful to retain the source proof, while developing the target proof, so that the user has the choice, depending on the application, of which extract algorithm to use.

2.3.3 Abstraction and Modification

Proof trees are internally represented within OYSTER as quite complex Prolog data-structures.[21] To avoid computational effort being expended on attempting to access individual semantic units the OYSTER representations of the proof trees are processed, by abstraction, into more accessible list structures called *rule-trees*. A typical rule-tree will either explicitly contain, or contain labels which allow for the direct accessing of, the following information:

- *Some* of the assumptions (hypotheses) made during the proof.
- The branching structure of the proof.
- The rules applied along with any corresponding arguments.
- An account of the dependencies between facts in the proof:

[21] Within the pre-processed OYSTER representation there are many Prolog variables hanging on to the various (sub)lists and it is generally hard to follow what parts of information form semantic units.

- dependency information concerning inter-relations between (sub)goals; and
- dependency information concerning inter-relations between (sub)goals and assumptions (hypotheses).

So, recalling the Curry-Howard isomorphism, the rule trees contain an account of the dependencies between facts involved in the computation of the λ-function constructed by the corresponding proof.

$$
\begin{array}{l}
(intro \text{ then } wfftacs) \text{ then } [elim(x, cv) \text{ then} \\
\quad [decide(v2 = 0 \; in \; nat) \text{ then} \\
\qquad [intro(s(0)) \text{ then } wfftacs \text{ then } apply(lemma\, 1), \\
\qquad decide(v2 = s(0) \; in \; nat) \text{ then} \\
\qquad\quad [intro(s(0)) \text{ then } wfftacs \text{ then } apply(lemma\, 2), \\
\qquad\quad (intro(v0 \; of \; pred(v2) + v0 \; of \; pred(pred(v2))) \text{ then} \\
\qquad\quad wfftacs) \text{ then } apply(lemma\, 3)]]]]
\end{array}
$$

Figure 2–6: The source rule tree for *Fibonacci*

As an example, and one which will play a prominent rôle when we come to discuss example transformations, we display, in **fig.** 2–6, a simplified representation of the rule-tree abstracted from the *Fibonacci course_of_values* proof (**fig.** 2–2). Each rule entry corresponds to a refinement application such that the above *rule tree representation* schematically corresponds to:

$$apply(Rule_1) \; then \; [apply(Rule_2) \; then [\; ... \; apply(Rule_n)] \; ... \;]],$$

and as such is akin both to a proof plan, which combines a number of proof tactics and/or rules into a large tactic such that a complete proof can be (re)produced from the plan (see next section), and to a skeleton of a proof in which the inference rules of the proof are recorded, but not the formulae to which they are applied.

The nesting pattern of the rule-tree list structure mirrors the branching pattern of the corresponding proof (the reader may wish to compare **fig.** 2–6 with **fig.** 2–3 of §2.2.6). This allows for the easy access, and subsequent modification, of individual (sub)proofs and (sub)branches during the course of the source to target transformation.

The main operational difference between, for example, the OMTS rule-trees and the proof tactics constructed by the CLAM proof planner is that the former are applied as they are constructed, rather than applying the plan upon completion. This has the advantage that any illegal mappings/transformations are detected as they occur.[22]

The General Rôle of the Rule Tree

We shall denote the refinement rule applications, represented within the rule-tree by R_{ref}. The information concerning hypotheses made during a proof and the inter-dependencies of the sub-goals will be denoted by R_{dep}.

The dependency information, R_{dep}, has to be abstracted from the rule-tree, which, in effect, retains a record of the inter-relations between *proof hypotheses* and sub-goals. For example, regarding **fig.** 2–2, the variable symbol $v2$ labels the *course_of_values* induction variable, and the variable symbol $v0$ labels the *course_of_values* induction hypothesis. Hence, the OMTS can determine from the construct within the rule-tree that constitutes the induction step witness,

$$intro(v0\ of\ pred(v2) + v0\ of\ pred(pred(v2))),$$

that an output for the recursive case depends on the outputs given by the induction hypothesis, $v0$, when the inputs are $pred(v2)$ and $pred(pred(v2))$ respectively. In other words, the dependent subsidiary function calls for $fib(n)$ are $fib(n-1)$ and $fib(n-2)$.

The rule-tree and sub-trees thereof are akin to large *tactics* in that they consist of arbitrary combinations of inference rules and proof tactics, such as quantifier elimination, case analysis application, and induction, by means of the pre-defined tactical *then*.

---

[22]However, for efficiencies sake, this approach is optional: if we are particularly confident that the transformations will not result in an invalid proof and therefore do not wish to waste time continually applying partial rule-trees then there is a mechanism setting which only applies the completed target rule-tree upon termination of the transformation.

An account of the dependencies between facts involved in the computation is represented within the rule-tree by recording the names of formulas to which inference rules are applied. So the rule-tree proof abstractions are, in effect, skeleton representations of proofs in which the inference rules of the proof, but not the formulas to which they are applied, are recorded.

A source proof is transformed by the application of *transformation tactics* to the source rule-tree. Hence the the transformation tactics themselves perform transformations on tactics (viz., the OMTS rule-trees). Using the transformation tactics, (sub)branches of the source proof can be accessed and the appropriate transformations made. They are called transformation *tactics* because it is their function to develop the target proof by modifying the source rule-tree according to certain pre (and post) conditions (see next section). The resulting target rule tree can then be applied at the OYSTER object-level.

The basic operations, of which the transformation tactics are composed, are designed to manipulate R_{ref} and may be categorized in a similar fashion to the *analogical transformation work* of Carbonell (Carbonell, 1983):[23]

Insertion: Object-level proof refinements are inserted into R_{ref}.

Deletion: Object-level proof refinements are deleted from R_{ref}.

Splicing: A portion of R_{ref} is spliced and mapped onto the target (possibly after the application of the other operations).

Concatenation: Separate portions of R_{ref} are joined to form a single target subsequence of rule applications.

Substitution: Source terms within R_{ref} are substituted for new target terms.

---

[23]In earlier research, the author has both reconstructed and extended Carbonell's *analogical transformation system*, (Madden, 1985), and investigated applying analogical transformation techniques to NUPRL proofs (Madden, 1987a).

The information R_{dep} is responsible for *guiding* the application of the basic operations.

From these basic operations, the transformation tactics can be categorized by more global functions such as :

- *Source to target induction schema transformation.*
- *Tupling transformations.*
- *Partial evaluation of (sub)proofs.*
- *Pruning transformations (or case analysis transformation).*

Tactic Transformation: Conditionally Guided Rule-Tree Modification

The OMTS transformations are, then, akin to tactic transformations guided in part by whether or not certain syntactic properties are true of the source rule-trees. Such syntactic properties function as transformation tactic pre-conditions. We can also predict the probable outcome of the application of a transformation tactic in terms of syntactic properties of the target rule-tree. A source to target transformation will be deemed successful if the target rule tree satisfies the post-conditions.[24]

The pre- and post-conditions for the induction schema transformations are fairly straightforward. For example, transformations from an exponential procedure to a linear procedure include, amongst their pre-conditions, that the dominant induction in the proof is a *course_of_values* induction (i.e the rule tree must contain a *cv_ind* construct). Amongst the post-conditions will be the presence of a *stepwise* construct in the target rule-tree. The details of other pre- and

[24]If the source rule-tree satisfies the pre-conditions then only in exceptional cases will a complete target rule tree be produced which violates the post-conditions.

post-conditions, such as those for the tupling transformations and the specialization pruning transformations are described in the appropriate chapters (especially *Chapters 4* and *5*).

Why Use Rule-Trees Rather than (a) the Proofs, or (b) the Programs

(a) Proofs will contain large amounts of information which is irrelevant to *both* execution and the optimizi ng transformations. Hence inefficiency would result from this additional information being subject to extensive manipulation in the course of the transformations. Conversely, the extract terms (or simply programs) have had much of the information relevant to the transformations abstracted away, rendering any (automatic) optimization more problematic. So the rule-trees serve as a good compromise, containing just the right amount of information, in particular dependency information, for transformation *and* for (automatically) constructing the associated proof.

(b) Given the proof specification, a rule-tree, unlike extract programs, will contain *all* the information required either to faithfully reproduce the *complete* proof from which it is abstracted, or to produce the target proof once transformations have been applied. In other words, rule-trees contain the information required to reproduce both the synthesis and verification components of a proof. This means that we can transform the rule-tree and then produce a target proof by applying the rule-tree to the target specification. In this way we ensure that the target extract program is correct with respect to it's specification, since it is extracted from a complete proof of that specification.

(c) Since the rule-trees are large tactics which can be applied at the OYSTER object-level to produce a complete proof, the OMTS optimization is tantamount to tactic transformation. The existence of pre- and post-conditions for the tactic (rule-tree) transformations cuts down considerably on the search space associated with constructing the target proof at the object-level. That is, a source rule-tree, together with a satisfied set of pre-conditions and a predicted set of post conditions, function as proof-plans for guiding the target construction through

the object-level search space. The transformation space is, then, equivalent to a planning search space in that tactics are selected for application according to whether or not pre-(post-)conditions are met. As such, the transformation space is far smaller than the object-level search space – in the order of 10^{15} times smaller – since each tactic combines many of the object-level inferencew rules. [25]

The OMTS exploits this property, together with further control factors such as various heuristics and the structure of the source proof (*Chapter 6*), to reduce further the amount of search involved in guiding the target construction. These and further issues concerning search and control of the transformations are elaborated upon in subsequent chapters (see especially §5.4).

The General OMTS Strategy

Letting ⊙ represent the *current node* in the developing target proof, and letting R be shorthand for "rule-tree", then, simplifying somewhat, the transformation can be schematically depicted by **fig.** 2–7 below. Both the *optimization* and *specialization* of programs through proof transformation require the *same* central transformation system module – the same basic transformation strategy depicted

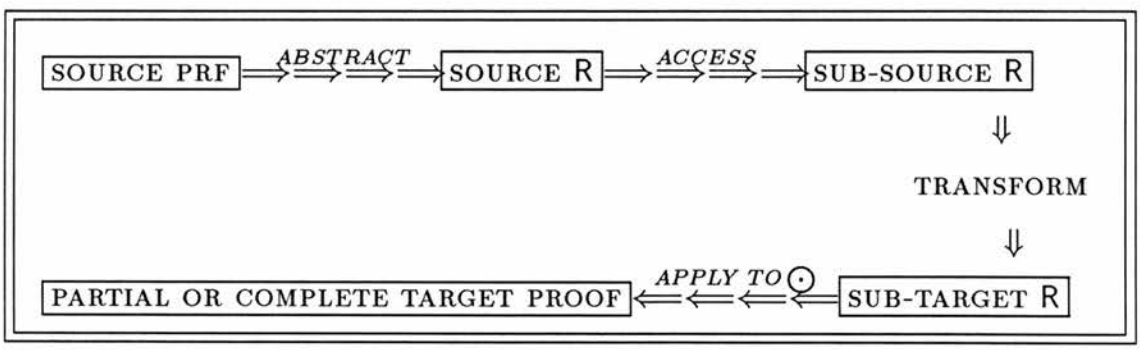


Figure 2–7: The OMTS transformation process

---

[25] The figure 10^{15} is only a rough average estimate, gleaned from a much more thorough account of the relative sizes of object level search spaces and the planning search spaces associated with numerous different proof plans produced by the CLAM proof planner. The reader should consult (Bundy *et al*, 1991) for the details.

in **fig.** 2–7 above – for their realization. This was an intentional design factor which greatly reduces the size (complexity) of the OMTS (primarily since it prevents duplication of large portions of code). It also means there is the potential to first *specialize*, or *adapt*, a proof and *then optimize* the specialized proof (or *vice-versa* whereby the *optimized* proof is then *specialized*).

Two Kinds of Recursive Program Optimization

There are two ways in which the OMTS can optimize a *recursive* source program by transforming the *induction* schema employed in the corresponding synthesis proof.

1. **Transformation of induction schemas:** The source induction schema is replaced by a different, but logically equivalent, target induction schema.[26]

2. **Transformation of induction cases:** The *step* and/or *base* cases of the source induction are transformed whilst retaining the same induction *schema* in the target.

We shall consider each of 1 and 2 in turn.

- *Transformation of induction schemas*

Source to target transformations of the *first* kind will transform the *way the source extract algorithm recurses on its input.* **Fig.** 1–3 and **fig.** 1–4 of §1.3.2 allow the reader to compare the *exponential* recursive behaviour of a source algorithm which computes the *Fibonacci* function with the *linear* behaviour of a target algorithm which computes the same function. Although the individual syntheses have much

[26]By logically equivalent induction schemas we mean that the associated induction theorems are inter-derivable. In *Appendix 1* we show, as an example, the logical equivalence of two of the most common induction theorems. This guarantees that any two proofs satisfying the same complete specification but differing only in which of the two schemas employed are *functionally equivalent.*

in common, in particular the general shape exhibited by the majority of inductive proofs, §2.2.5, the main difference between the source and target proofs is that the former uses *course_of_values* induction where as the latter uses *stepwise* induction.

In general, the main points where inductive proofs diverge corresponds to:

(i) the choice of induction schema employed and

(ii) the type of object introduced at the induction step, and

(iii) how the object is witnessed (instantiated) and

(iv) the subsequent verification of the instantiated schema

By incorporating general rules that associate (i) and (ii) with the kind of recursive behaviour exhibited by the target algorithm, and utilizing the source definitions and dependency information in the source proofs to achieve (iii), the OMTS is able to construct the target proofs automatically.

Although the verification component, (iv), will differ from proof to proof, the verification strategy invariably follows the same procedure of applying refinement rules that primarily consist of unfolding the recursive branches with the equational definitions that define the function computed by the extract program.

• *Transformation of induction cases*

Transformations on induction cases correspond to transforming the sub-proofs of the base and /or step case sub-goals *without altering the particular schema for which the sub-goals are cases.*

Different recursive behaviour can be induced in algorithms, satisfying the same specification, by refining the step and base cases of the *same* schema in different ways.

As far as the OMTS optimizations are concerned, we shall be *primarily* concerned with transformations of the first kind (*Chapters 5*). That is, transformations that replace, for example, the following *stepwise* induction proof construction

with a completely different one, as opposed to just transforming the constructs within the boxes:

$$\lambda x.\, p\_ind(x, \boxed{\phi_0}, [x', \phi_{H_{ind}}, \boxed{\phi_C}]).$$

However, one particular kind of source to target proof transformation on induction cases with which we shall be concerned is *the transformation of nested inductions.* Nested inductions are often employed when synthesizing *auxiliary recursive functions*, that is, functions which in computing a self-recursive call must appeal to some other function, either directly or indirectly.[27]

A nested induction may lead to inefficiency since for each of the recursive passes induced by the outermost induction, the program will have to fully recurse on the innermost recursive schema induced by the innermost induction. Thus the average time efficiency of such programs will be a multiple of the time efficiencys associated with the two inductions. So for example, the recursive definition of the following schematic function f:

$$f_1(n) \;=\; h(f_2(n), f_1(n-1))$$

contains both an auxiliary function call, $f_2(n)$, and a self-recursive call, $f_1(n-1)$. Each time a recursive call is made on f_1, the program must fully recurse down the schema associated with f_2. Proofs wherein a nested induction is applied at the step case of the outermost induction may, for example, yield a program construction of the following form:

$$\lambda x.\, p\_ind(x, \phi_0, [x', \phi_{H_{ind}}, \lambda x'.\, p\_ind(x', \phi_0, [x'', \phi'_{H_{ind}}, \phi_C])]),$$

where in order to evaluate the step case of the outermost *stepwise* induction, p_ind, on x, with induction variable x', the program must evaluate a nested induction

---

[27]The nesting may be indirect since the structure introduced for the step case of an induction may well correspond to the application of an extract term from another proof which itself employs stepwise/list induction (which may in turn incorporate nested or unnested induction schemas).

on x'. Optimizations on such extract terms are performed through "merging" the innermost induction with the outermost induction. This is achieved by introducing a tuple structure at the cases of the outermost, and only, target induction which tabulates the computation associated with the innermost source induction. Details of this process are provided in §5.3.6.

Two Kinds of Specialization Transformations

In addition to the automatic, and correctness preserving, optimization of recursive programs, the OMTS is also capable of adapting, or specializing, programs. I.e., the OMTS is capable of automatically reacting to a change in a source program specification that corresponds to a partial evaluation on some input parameter.

1. **The specialization of a proof by cases:** proof case splits are subject to pruning transformations guided by *redundancy information* abstracted from proof hypothesis lists.

2. **The specialization of induction schemas:** proof induction schemas are pruned following partial evaluation.

We shall again consider 1 and 2 in turn.

• *Pruning of proof case splits*

As discussed in §2.2.10 the introductory account of *Chapter* 1, a partially evaluated proof may exhibit redundancies, particularly with respect to nested case split proof constructs. The reasons for this will be explained in the relevant chapter (Chapter 5). We shall however expand on the introductory account of specialization by saying a little concerning the effects that pruning specific proof constructs has on the associated program constructs.

The specialization of programs through pruning transformations performed on proofs will, for example, transform the *nat_eq* proof construction

$$nat\_eq(x, y, \phi_{x=y}, \phi_{(x=y)\rightarrow void})$$

associated with the refinement application $decide(x = y\ in\ nat, new[h])$, into a target construction corresponding to either of $\phi_{x=y}$ or $\phi_{(x=y)\rightarrow void}$. Exactly which disjunct is entered as a target proof assumption depends on the following.

(i) Whether or not $x = y$ is true under the specific partial evaluation of x and/or y.

(ii) Whether or not either of the (sub)proofs, $P_{\phi_{x=y}}$ or $P_{\phi_{x=y\rightarrow void}}$ associated with $\phi_{x=y}$ or $\phi_{(x=y)\rightarrow void}$ are *redundant* under the specific partial evaluation of x and/or y (as far as computing the root node specification is concerned).

So (i) and (ii) encapsulate the pruning transformation pre-conditions, and the resulting target constructs the post-conditions. Pruning transformations guided by considerations relating to (i) are dubbed *normalization pruning*, and those guided by considerations relating to (ii) are dubbed *dependency pruning.*

By way of example, suppose we have a nested case split structure where the outermost case split is determined by case conditions $x = z \vee (x = z) \rightarrow void$, and the innermost case split is determined by case conditions $x = y \vee (x = y) \rightarrow void$, and the nesting corresponds to the following conditional form C:

C: $if\ \ x = z\ \ then\ \phi_{x=z}\ \ else\ \ if\ \ x = y\ \ then\ \phi_{x=y}\ \ else\ \ \phi_{x=y\rightarrow void}.$

In our λ-calculus notation then, the source extract term, ϕ_{source}, will contain the following nested *nat_eq* construct S (§2.2.2):

$$S: \quad \phi_{source} \ = \ nat\_eq(x, z, \phi_{x=z}, nat\_eq(x, y, \phi_{x=y}, \phi_{(x=y)\rightarrow void})),$$

and the below diagram, **fig.** 2–8, schematically depicts the passage from the initial partial evaluation of the proof from which ϕ_{source} is extracted to the final target extract yielded by the dependency pruned proof. We assume that the partial evaluation is on y such that $x = y$ becomes true (or equivalently, $x = y \rightarrow void$ is false). This partial evaluation, or *intialization*, is denoted by the expression PARTIAL EVAL$(x \mapsto y)$. The pruning operations are represented by expressions of the form PRUNE$(P_{\phi_{case}})$, where P is the (sub)proof – or evidence – at case

condition CASE, corresponding to the extract construct ϕ_{case}, and where PRUNE is the pruning operation responsible for removing $P_{\phi_{case}}$. The labels S, N and D denote the source, normalized and dependency pruned programs respectively.

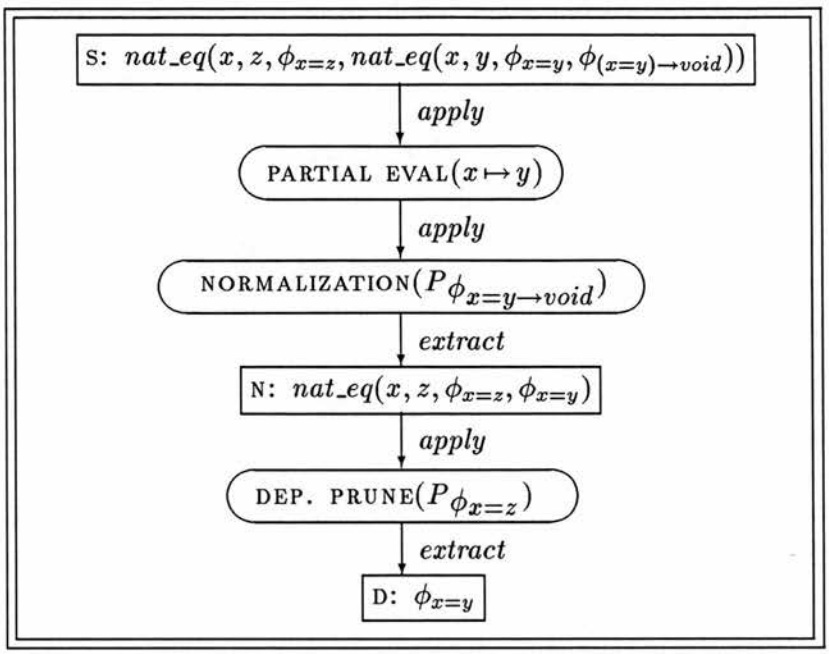


Figure 2–8: The effects on the proof/program constructions of specialization

The passage from S to N is fairly straight forward, once $x = y$ is known to be true, then we can prune the case split branch corresponding to $x = y \rightarrow void$. Such pruning can be unproblematically performed on the conditional form, C, of the algorithm yielding:

C': $if\ x = z\ then\ \phi_{x=z}\ else\ \phi_{x=y}$.

However, the formalization of the algorithm as a *constructive existence proof*, its subsequent partial evaluation and the application of normalization pruning may then allow C' to be *automatically* simplified, by the use of dependency pruning, to the expression:

C": $\phi_{x=y}$.

This is basically because dependency information, contained in the proof hypothesis lists, may tell us that if $x = y$ is true then the output, $\phi_{x=y}$, does not neces-

sarily depend on whether or not $x = z$, and hence the outer-most case split may be pruned. There is nothing in the conditional forms, C or C', which motivates such a transformation. In *chapter 4* we explain exactly how the extra information present in constructive proofs - *information which is not concerned with simple execution* - can be exploited to automatically achieve the transformation from N to D (or equivalently from C' to C"). We shall also give considerable attention to the following facts:

- Due to the design of the OMTS each stage of the specialization process, depicted in **fig.** 2–8 above, is correctness guaranteed. In particular, correctness of the target, with respect to the modified specification, is ensured. Usual program transformations do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour.

- *Both* N and D satisfy the *same* complete specification, despite the fact that N and D define different algorithms (an indication that, recalling the problem of refinements capture, although a complete specification may totally capture a programs input-output relation, it may not do so unambiguously).

In *Chapter 4* we describe and discuss the author's program, through proof, specialization system, which adapts programs in the broad manner described above. The specialization of a program through the partial evaluation of the initial specification, and corresponding synthesis proof, provides a good example of how proof transformation can assist in modifying a programs internal structure in accordance with an initial modification of its specification. Further detail concerning the rather subtle, and *novel*, properties of specialization, in particular regarding correctness and dependency pruning, will be discussed in that chapter.

- *Elimination of induction schemas through partial evaluation*

Source to target transformations of the third variety will, in general, alter the functionality of the source algorithm (or more precisely, the range of inputs for which the program will compute an output will be modified). This is done by

removing the source programs recursive behaviour altogether. Induction is employed in a proof when we wish to establish formally that a certain property is true of all instances of a specified data-type. In cases where we wish to establish that some property is true only of a (some) specific instance(s) of a specified data-type then we may partially evaluate an induction schema according to the specific instance(s).

In other words, rather than proving that if $A(0)$ and $A(v) \vdash A(s(v))$ that therefore $\forall x A(x)$, we may wish to prove that A holds for a specific value n, or that A holds for all objects with a value less than a specific value n: i.e. we may wish to *specialize* the induction argument. This is the case for the source to target *specialization* of *recursive* programs, wherein the recursion schema is removed from the extract by instantiating (partially evaluating) and then pruning the induction schema of the corresponding proof. This type of pruning we call *induction grounding.*

So, given a specialization of n, induction grounding replaces the infinite sequence of "sub-proofs":

$$P_{\phi_A(0)},\ P_{\phi_A(1)},\ P_{\phi_A(2)},\ P_{\phi_A(3)},\ \ldots,\ P_{\phi_A(n)},\ \ldots,\ P_{\phi_A(v)},\ P_{\phi_A(v+1)}, P_{\phi_A(v+2)},\ \ldots,$$

implicit within (or obtained by "unraveling") a proof of $A(v)$ by, for example, stepwise induction:

$$p\_ind(x, P_{\phi_A(0)}, [x', P_{\phi_A(v)}, P_{\phi_A(v+1)}]),$$

by a nested application of cuts which sequence into the proof the first n "sub-proofs" of the infinite sequence above.

2.4 Summary

We now collect together and summarize the main points of this chapter:

(1) *Proofs as Programs*: The basic principle behind the OYSTER approach to program synthesis is the Curry-Howard isomorphism, which allows us to relate programs with proofs in (a logic based on) Martin-Löf type theory. It gives the mathematical basis for developing a formal system to synthesize a program. The program extraction mechanism is based on constructive logic.

(2) OYSTER *Refinement*: We distinguished two main categories of λ-calculus refinement available to OYSTER: the *intro*duction and *elim*ination rules which operate, respectively, upon the hypotheses and conclusions of a goal. We also described the application of the *seq*, rule, lemma application, and, crucial for the synthesis of recursive programs, the various induction schemas available in OYSTER.

(3) *Induction is Dual to Recursion*: Regarding the latter, induction, we described how primitive recursive schemas permit the definition of primitive recursive functions in terms of the cases based on the type constructors. As such, there exists a duality between induction and the recursion of the OYSTER functional language.

(4) *Alternative Syntheses Yield Alternative Recursions*: Due to the properties of the OYSTER specification language, different proofs, and thereby programs, can be constructed from the *same* formal specification, depending on how the specification is refined. In particular, the key choices and decisions that determine the nature of the *recursive* process generated by a program correspond to:

(i) which induction schema to employ;

(ii) the choice of the induction candidate;

(iii) the (constructive) type of object introduced at the induction cases;

(iv) how to witness the subsequent cases; and

(v) which definitions are appealed to in order to complete the verification component of the proof.

The procedural commitments stemming from such choices have a marked bearing on the efficiency with which a recursive program computes the specified input/output relation.

(5) *Correctness*: Furthermore, since a program extracted from a complete proof is necessarily correct with respect to the specification embodied in the root node of that proof, then each algorithm synthesized from the same specification, by making alternative choices during the refinement, is guaranteed correct with respect to that specification.

(6) *Typical Inductive Proof Plan*: We discussed the typical strategy, or structure, of the majority of inductive proofs, and, by way of example, provided alternative syntheses of a program for computing the *Fibonacci* sequence of natural numbers. These examples indicted how the above choices effect the program construction, and familiarized the reader with OYSTER synthesis and notational conventions.

(7) *Efficiency and Refinements Capture*: We also indicated, with the *Fibonacci* syntheses as cases in point, that the most natural way of synthesizing a (recursive) program is by no means the way of attaining the most efficient program. Indeed, on empirical evidence alone, there appears to be an inverse relation between, on the one hand, the complexity of the recursive process generated by an extract, and on the other, the complexity of the proof by (from) which it was constructed (extracted).

Relatedly, we made a plea for more research into the problem of refinements capture: the problem of determining whether or not a specification totally and unambiguously captures the *desired* program input-output relation, and captures it in such a way that the output is computed efficiently from the input.

(8) *Proofs Contain More Information than Programs*: We identified an important feature of the OYSTER sequent calculus as being the recording, at any stage (node)

during a proof development, of all the dependencies (assumptions and hypotheses) required to complete that proof stage within a hypothesis list. Due to the Curry-Howard isomorphism, such dependencies represent a record of the dependencies between facts involved the computation. Proofs also contain information concerned with verifying that the extract term computes the task described by the specification. Finally, proofs contain a specification. Not only does this afford us with a correctness guarantee and a termination condition, but it also allows for transformations that modify input/output behaviour.

(9) OMTS Inherits Properties from OYSTER: We drew particular attention to how properties of the OMTS are inherited from the (object-level) refinement system. We discussed optimization and specialization in terms of the effects transformations have on the proof constructions.

(10) *Control, Correctness, Automatability, and Generality*: The fact that alternative procedural commitments – corresponding to (i) to (v) of (4) above – can be correlated with the efficiency of the resulting (sub)computations, forms the backbone of OMTS design: programs are optimized through the application of *transformation tactics* to modify the procedural commitments made during a source proof synthesis. Considerations directly relating to (4),(5),(6),(8), and (10) above afford the transformations with a control mechanism, and play a key role in ensuring that the OMTS satisfies the main desirable criteria for a transformation system: automatability, correctness, and generality of design.

(11) *Rule-Tree Abstractions*: We outlined the main applications of the OMTS, broadly categorized as *recursive program optimization* and specialization. We also provide a high-level description of the OMTS design, the key features being the abstraction of rule-trees (or proof-plans), from the proofs, which are then subject to mapping and/or transformation rules. The rule-trees allow for efficient processing, whilst containing all the information required for the proof modifications.

(12) *The Main* OMTS *Transformations*: We categorized the main kinds of transformation that we shall cover in this thesis:

- Recursive program optimization through transformation of induction schemas.
- The specialization of a proof by cases:
  - Initialization (partial evaluation);
  - Normalization; and
  - Dependency pruning
- The specialization of induction schemas (induction grounding).

Chapter 3

A Review of Work Relating to Program Transformation

3.1 Introduction

The body of work, in the available literature, relating to program transformation is fairly extensive. In this chapter we therefore limit ourselves to reviewing briefly some of the more influential transformation systems that are either similar, in some sense, to the techniques employed in the OMTS, or provide interesting alternatives. These will include:

- the fold/unfold strategy;
- the tupling technique;
- the partial evaluation of programs; and
- program through proof transformation.

The first of these, the *fold/unfold strategy*, features in the majority of the systems reviewed. The strategy originates from (Burstall & Darlington, 1977b). The general idea is to transform an inefficient, source functional program into an equivalent, more efficient, target functional program through a process of unfolding and folding recursive definitions. The target program is defined in terms of the source program and then the fold/unfold process is used, as a re-writing strategy, to derive a recursive definition for it independent of the source program. The large

search space associated with this process offers ample scope for the employment of various (heuristic) control strategies. Examples, together with formal definitions of *folding*, *unfolding* and further transformation rules, are provided in §3.2.1.

We shall also be concerned with certain properties that are generally considered desirable criteria of a transformation system, and to what degree the systems reviewed measure up to these criteria. They are:

- the correctness of the transformations;
- (the degree of) automatability;
- the generality of the transformations; and
- the expressiveness of the program specification language.

3.1.1 The Specification Language and Preserving Equivalence (or Ensuring Correctness)

Before embarking on the systems review, we shall provide a brief introductory commentary on the aforementioned criteria. We divide our discussion into considerations concerning the direct transformation of executable code (programs), and considerations concerning the transformation of programs through proof transformations.

Transformation of Programs

Although a desirable property, not all program transformation systems are primarily concerned with ensuring the correctness of their transformations. Such systems are referred to as *heuristic* systems, and rely generally on heuristic re-write rules which will produce a target program without the considerable extra work required to formally establish that it is equivalent to the source program.[1] Examples of

[1]Such systems should not be equated with systems that employ heuristics to *select* appropriate re-write rules (i.e., to control the path through the transformation search

such heuristic systems include the LOPS system (Bibel & Hörning, 1984), Grant and Zhang's "list-processing" optimization system (Grant & Zhang, 1988) and the original implementations of Darlington's *fold/unfold* strategy (Burstall & Darlington, 1977b). [2] Later extensions to (Burstall & Darlington, 1977b) are provide with a partial correctness guarantee.

- *Providing Correctness Criteria*

Establishing that source to target transformations are *totally* correctness preserving is a broader issue than establishing the correctness, or equivalence, of the various transformation re-writes employed during the transformation. *Total* correctness includes, in addition to correctness, establishing that the transformations will terminate.

In *general*, and due to undecidability factors, it is not possible to establish total correctness unless some sort of restrictions are placed either on the form of the re-write rule applications, or on the sub-set of logic within which the (executable) specifications are formalized.[3]

For example, both (Kott, 1978) and (Tamaki & Sato, 1984) have the restriction that there are always an equal, or greater, number of unfold steps than fold steps. In this way termination, and correctness, can be established ((Tamaki & Sato, 1984) is discussed in §3.2.1).

---

space). Although heuristics may be responsible for selecting appropriate re-write rules, whether or not the system is correctness preserving depends on whether or not the rules themselves are equivalence preserving.

[2]The LOPS system is in fact more akin to an interactive synthesis system for developing logic programs from an executable specification. The LOPS system is compared with the theorem proving approach to synthesis (namely NUPRL synthesis) in (Madden, 1988c).

[3]The reason for the undecidability is essentially due to the possible non-termination of a target program produced by the *arbitrary* application of folding. For example, although the identity function, $id(x) = x$, is terminating, a non-terminating target can be obtained by folding the definition against itself to produce $id(x) = id(x)$.

A similar approach is taken by (Scherlis, 1980) where folding is restricted to those newly defined functions wherein at least one (subsidiary) function call has been unfolded. Without such restrictions total correctness cannot be ensured.

For the most part, we shall be primarily concerned with whether or not systems have the property of correctness (since, without some kind of restrictions, non-termination will always be a fact of computational life). That is, *if* a source to target transformation terminates *then* does it terminate with a correctness guaranteed target program.

Of those systems where correctness is a primary goal there is generally one main approach, the refinement of executable specifications (viz. programs) using equivalence preserving re-write rules. The intention is that a source program is transformed by representing it within an executable subset of some logic, and then applying re-write rules that ensure that the input/output relation specified in the source program remains unchanged.

The motivation behind such systems is that each individual re-writing of the source program/specification is in itself guaranteed to preserve equivalence (given the re-write/logic sub-set restrictions). In this way correctness is ensured by the actual (target) program construction process, hence removing any need to (directly) provide lengthy equivalence proofs (of the source and target programs). In the case of recursive program transformation, this means that there is no (direct) recourse to lengthy inductive proofs to establish the correctness of the transformations.

However, this approach somewhat shifts the problem of providing correctness guarantees to the program construction process itself: that the re-write rules are in themselves correctness (equivalence) preserving needs to be established, and this will, as a general rule, require as much effort as providing an explicit proof of correctness for the source to target transformations.

For example, many of the systems that employ the *unfold/fold* strategy re-write the recursive step(s) of a source program through the application of various *equality* lemmas, each of which needs to be proved (by induction) if the source to target

transformation is to preserve equivalence (Gregory, 1980; Manna & Waldinger, 1980; Tamaki & Sato, 1984) (we briefly describe these systems in §3.2.1).

Hogger and Clark, whose work we also address in §3.2.1, place the condition on their source to target transformations that the input/output relation defined by the executable specification is a sub-set of the relation computed by the target program (Hogger, 1981; Clark, 1977). That this condition is met, again, requires lengthy inductive proofs.

Furthermore, with such systems, any extension to either the sub-set of logic within which the programs are formalized, or to the set of re-write rules (or both) will require a corresponding extension to the equivalence proof(s).

A more arduous approach is to provide termination proofs at the end of each source to target transformation: terminating programs are then deemed valid since they could only have arisen from equivalence preserving re-writing. For recursive program transformations, these termination proofs will require induction. Unfortunately, termination proofs are generally an undecidable problem, so this approach, again, only ensures partial correctness.

- *The Specification Language*

Kowalski's famous slogan, ALGORITHM = LOGIC + CONTROL, succinctly conveys the notion that, *ideally*, the logic component of a program should be a clear, and correct, statement of the problem, while the control component should be distinct from the logic component and responsible for the efficiency of the program (Kowalski, 1979).

In practice, it is not generally possible to attain a totally clear and distinct separation of the two components of an algorithm, and the degree of autonomy between the two components is generally dependent on the nature of the specification language.

In general, regarding program transformation systems, the specification language and the programming language are required to be virtually one and the

same, since a source program is optimized through the direct application of rewrites to that program. A drawback of this approach is that since the specification itself is tantamount to an executable (source) program then it becomes difficult to avoid placing constraints, *within that specification*, on how the program is executed. This is particularly the case with systems that use a simple functional programming language where the procedural content of the specification/program to be refined may determine to some degree *how* the resulting target program computes the specified input/output relation. This is clearly a restriction since, ideally, how the program computes it's output should be determined by how it is constructed from it's specification, and not by the specification itself. Or, in other words, for the purposes of transformation, the specification should have a minimal effect on the dependencies between facts involved in the computation of that specification.

On the other side of the coin, there is the problem of refinements capture (*cf.* §2.2.9): how well can the specification language be used to capture the description of the task to be computed. Some researchers have found that using a specificatuion language that is identical to the target language is too restrictive (i.e., it becomes very difficult to specify the exact computational task at hand, and to separate it's description from control issues). Hence the development of special purpose specification languages which facilitate the problem description. For example, at Imperial College a functional language is being developed as a successor to HOPE$^+$ (Darlington, 1989). The language includes facillities for logic, constraint and object-oriented programming features. As such, it allows for the formalization of high-level specifications, and the functional features support the the transformational development of HOPE$^+$ like programs. A further example is provided by Manna and Waldinger who have developed a "flexible" specification language for their SYNSYS system, §2.1.7, which can, to some extent, be tuned to the users requirements (Manna & Waldinger, 1980).

• *Automation and Generality*

Practically all the fold/unfold systems rely on user interaction. This applies equally to systems that are primarily concerned with synthesis, as opposed to optimization, and which use the *fold/unfold* technique such as Bibel and Hörnigs *Heuristic Program Construction System.* Of those systems that have achieved some success in (partially) automating source to target fold/unfold transformations, considerable reliance is often placed on some form of user-provided control program. This is the case with Sato and Tamaki's logic transformation system, (Tamaki & Sato, 1984), and Darlington's current HOPE$^+$ system (Darlington, 1989).[4]

In general, the larger the class of transformations desired, then the greater the number of transformation rules that the system must have access to, and hence the greater the search space associated with the (legal) re-writing of the source code. So the more general the system, the more difficult is the task of automating and/or controlling the search within the transformation search space.

An interesting approach to avoiding such control problems is to make the specification language flexible such that it can be tailored to a particular function's requirements. (This strategy is adopted in (Manna & Waldinger, 1980)).

Another approach is to employ some form of meta-language to specify tactics, with pre- and post-conditions, which can then control the object-level re-writing. (This strategy is adopted in (Feather, 1979a; Green, 1991)).

Chin describes how an impressive degree of automation can be obtained for fold/unfold transformations that employ the *tupling technique* (Chin, 1990): the re-writing of a source program is guided by redundancy information culled from an (automatic) construction and analysis of *dependency graphs.* In this way the dependencies between facts involved in the source computation are rendered open for inspection and modification such that repeated sub-computations are then grouped together into a single, more efficient, target tuple structure.

---

[4] HOPE$^+$ is an extension of NPL with built in tupling procedures (Burstall & Darlington, 1977b).

We shall discuss all the aforementioned references in §3.2.1.

Transformation of Refinement Proofs from (non-executable) Specifications

An alternative, and altogether different, approach to program transformation is that adopted by the author and already outlined in *Chapters 1* and *2*: program transformation through proof transformation. With this approach, if the termination condition is met – the production of a complete target proof – then the transformation is *ipsofacto* correctness guaranteed.

Following on from the previous section, ideally, as far as transformation is concerned, we would like a situation wherein we have access to fairly clear and distinct representations of:

- a description of the task being performed (a declarative specification);
- a synthesis component (incorporating an account of the dependencies between facts involved in the computation); and
- a verification component (i.e., a verification of the method).

As discussed in *Chapter 2*, formal proofs of (possibly non-executable) program specifications (such as those synthesized within the OYSTER system) go a considerable way toward offering such distinctive criteria. Hence the motivations behind the approach of treating the existence (synthesis) proofs, resulting from the refinement of formal (and non-executable) specifications, as the objects of transformation.

Although the formal proofs of correctness and existence for the source program may be rather lengthy, transformations performed on such proofs have two advantages: firstly, the correctness guarantee of the source to target transformations; and secondly, much of the extra information in such proofs, superfluous to the actual computation, can be exploited for the transformations (particularly with respect to automation).

The account of the dependencies between facts involved in the computation, corresponding to the dependencies between facts involved in the proof, offers precisely the kind of information we wish to exploit for the purposes of (automatic) transformation. We illustrated one means of exploiting this extra information in our introductory account of *specialization* (§2.3): the adaptation of programs to special situations through (pruning) transformations performed on existence proofs. The general specialization methodology originated from (Goad, 1980b; Goad, 1980a), and we provide in §3.2.3 an account of the main properties of Goad's specialization system. Due to the design of Goad's system, the property that all extract programs are correct with respect to the target specification is not directly exploited.

As far as the author is aware, apart from Goad, no other implemented systems which perform program transformation through proof transformation exist. We shall, however, provide a brief overview, §3.2.3, of a suggested proof transformation system design, (Pfenning, 1988), which although it differs from the OMTS design, does (or rather could) exploit the correctness properties of existence proofs.

3.2 Program Transformation Review

The lay-out of the review is as follows:

§3.2.1 We begin by describing the *fold/unfold* strategy for program transformation, within the context of Darlington's pioneering NPL program transformation system. We give formal definitions of *folding* and *unfolding* and provide a worked example of program transformation by the fold/unfold strategy. Of particular relevance to subsequent chapters are the following:

- lemma introduction (required to invent new procedures such that folding can occur), and
- the *tupling technique* used to group, or merge, together separate recursive expressions into a single function call.

We then provide brief surveys of systems which employ the fold/unfold strategy in some guise or another. The review of (Chin, 1990) provides more detail on the tupling technique, particularly regarding it's automation.

§3.2.2 We then move on to review briefly systems that use *partial evaluation*, or *explanation based learning* techniques, in order to optimize a source program by, in some sense, observing its behaviour when run on a concrete, or abstract, example.

§3.2.3 Finally, we turn to program through proof transformation. We review Goad's work on program specialization, and briefly describe a suggested methodology toward proof transformation (Pfenning, 1988). As already mentioned, there is little to review concerning this approach (especially regarding any working implementation), and so for a detailed account the reader must wait until the descriptions of the author's OMTS system.

Such a categorization of systems is primarily for presentation purposes, and should not be taken too rigidly since, for example, most of the systems that employ partial evaluation also employ the fold/unfold technique, and Goad's proof transformations are initialized by partial evaluation.

3.2.1 The *Fold/Unfold* Strategy

The most influential application of the *fold/unfold* technique is within Darlington's NPL program transformation system. Many of the more recent system designs are based upon Darlington's framework for refining clear but inefficient programs (or executable specifications) into their efficient equivalents (Gregory, 1980; Manna & Waldinger, 1980; Tamaki & Sato, 1984).

The fold/unfold technique is a specific kind of re-writing which involves matching, and replacing, recursive terms from the developing branches of the target program: by a process of re-writing recursive definitions, a recursive definition for the target program is derived which is independent of the source definition. Although it has been employed, in various guises, in many of the existing transformation

systems (and suggested system designs), it originated within the NPL context of developing and optimizing simple functional programs.

The fold/unfold technique is usually initiated by some form of *eureka step* where the desired target program specification is defined, via lemma introduction, in terms of the source, thus setting the scene for unfolding, followed by folding, to take place. The generation of such lemmas has proved notoriously difficult to automate, as have the control issues involved in deciding whether or not to introduce a fold, or to continue unfolding.

Darlington's NPL Functional Program Transformation System

Darlington and Burstall have designed a research tool for the development of program transformation methodologies. Program transformations are defined as schematic re-writing systems within a functional programming language, together with constraints on the instances of the schemas that must be met in order for the transformation to be valid. The system relies heavily on user instantiations together with rules for their evaluation.

Darlington's NPL transformations rely heavily on symbolic evaluation and are achieved mainly by sequences of *foldings* and *unfoldings* together with *instantiations* provided by the user. In all, there are six main *transformation rules* (re-write rules), R1 to R6, the first two of which, *unfolding* and *folding*, are defined as follows:

(R1) *unfolding*: If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E, replace it by the corresponding instance of E' obtaining F'', then add the equation $F = F''$.

(R2) *folding*: If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F'', then add the equation $F = F''$.

The central strategy of the *fold/unfold* transformations consists of generating lemmas to introduce recursions into the developing target program by application of the above *folding* rule R2.

The third transformation rule is composed of numerous *Laws* such as those for associativity, commutativity, etc.

(R3) *laws*: $X+Y=Y+X,\ X+(Y+Z)=(X+Y)+Z,\ X\times Y=Y\times X$ *etc.*

An important facility is the introduction of a *where* clause, R4, by deriving from a previous equation $E = E'$, and given equations $F_1, ...F_n$, a new equation thus:

(R4) *Where Clause Introduction Rule (abstraction)*: $E = E'[u_1/F_1, ..., u_n/F_n]$ *where* $\langle u_1, ..., u_n\rangle = \langle F_1, ...F_n\rangle$.

So abstraction consists of replacing parts of an expression, in the body of an equation, by variables, and then defining these variables in a *where* clause. This introduction of *where* clauses is an essential part of evaluating the recursive branches of the *target* program.

Finally, NPL employs a *definition rule*, R5 and an *instantiation rule*, R6.

(R5) *Definition*: Define a new target recursive equation in terms of the source recursive step.

(R6) *Instantiation*: Create a substitution instance of an existing recursion equation.

Darlington's system is very much interactive, the onus of the optimization, or source-to-target-correctness, §2.2.9, falling on the user. She or he must provide the following instantiations for the functions she or he wishes to improve:

- the inefficient ("naive") version of the program (i.e., the source);
- the instantiated left hand side of the target recursion schema base equation; and
- the instantiated left hand side of the target recursion schema step equations.

The system then proceeds to evaluate the base and recursive branches for the efficient program as follows:

Base case: Armed with the above instantiations the system selects an equation, instantiates it as the user requests and then unfolds the right hand side.

Step case: The system proceeds as above except that the unfolding may be followed by the application of *laws*, R3, then by sequences of foldings.

Folding often consists of simple matching operations, searching through the current equation list, consisting of the originally provided equations together with those developed so far, and finding an equation an instance of whose right hand side occurs within the right hand side of the developing equation. However, it is the guided control of folding that can lead to more interesting behaviour: different kinds of folding can lead to different recursive patterns and, in particular, *forced folding* uses information from a failure to achieve a simple fold to direct the development of the equation so that a fold can be achieved. So the outcome of the optimization can be determined by the control of folding.

Finding a suitable sequence of unfoldings, finding a suitable fold, and the decisions associated with when to stop unfolding and to introduce a forced fold, all contribute toward search and control problems. A *pre-set effort bound* prevents the repeated application of unfolding becoming too deep.

• *Example: Linearization of Fibonacci by Tupling Followed by Fold/Unfold*

Tupling is an important means of linearizing exponential procedures. It works by grouping together, in a single recursive tuple function, the separate recursive expressions in the source procedure. So, with $i \geq 2$ the "conditions" for tupling are as follows:

Condition 1: There exist two or more *recursive calls* (or expressions), $f(n), ..., f(n-i)$, which share some *common recursion variable(s)* in a function definition.

Condition 2: There exists a fixed sized tuple - the *eureka tuple* - within which common subsidiary recursive calls arising from the execution of each of

$f(n), ..., f(n-i)$ can be merged, thus forming a recursive function without the original redundancy.

The provision of the fixed sized tuple constitutes the *eureka step* for program transformation by tupling. In most systems that employ tupling, or some similar form of tabulation, it is achieved through some variant of the abstraction rule (R4). It is also generally achieved through considerable user interaction.

The transformation process starts with the *source Fibonacci* proof of the previous section, duplicated below:

$$
\begin{array}{lll}
(1)\ fib(0) & = & 1; \\
(2)\ fib(1) & = & 1; \\
(3)\ fib(n+2) & = & fib(n+1) + fib(n).
\end{array}
$$

Note how this definition satisfies **condition 1** above.

The process of defining the desired optimization in terms of the "course of values" definition, as described in the previous section, is what Darlington refers to as the "key to the optimization", or the *eureka step.* This is done by the introduction of the auxiliary function fib_{tup} (thus satisfying condition 2 above):

$$(4)\ fib_{tup}(n) = \langle fib(n+1), fib(n) \rangle.$$

This auxiliary function acts as a tuple which, in effect, replaces the source recursion schema with a target schema which combines identical recursive calls.

So Darlington's strategy is motivated by the observation that significant optimization of a (declarative) program generally implies the use of a new recursion schema. This process depends on the *user* providing the requisite definition of the *eureka* tuple fib_{tup}.

The system proceeds to evaluate the recursive branches of the auxiliary function, given the original equations and the instantiated base cases. Armed with the original equations, it is a simple matter for the system to evaluate the base case for fib_{tup} given the left hand side of the equation, $fib_{tup}(0)$,

(5) $fib_{tup}(0) = \langle 1, 1 \rangle$.

By using R4 to introduce a *where* clause, the system produces a definition of Fibonacci, *in terms of our auxiliary function* fib_{tup},

(6) $fib(n+2) = (u1 + u2)$, where $\langle u1, u2 \rangle = fib_{tup}(n)$.

Forced folding then comes into play for the optimization of $fib_{tup}(n+1)$: given the instantiated left hand side of the recursive step, unfolding produces the equation

(7) $fib_{tup}(n+1) = \langle fib(n+1) + fib(n), fib(n+1) \rangle$.

The system then attempts to fold this equation with

(8) $fib_{tup}(n) = \langle fib(n+1), fib(n) \rangle$,

but fails since there is no direct match between the two. By observing that all the components necessary to match equation (8) are present within equation (7) the system *forces* the match, by using R4, to rearrange equation (7) to the following

(9) $fib_{tup}(n+1) = \langle u1 + u2, u1 \rangle$, where $\langle u1, u2 \rangle = \langle fib(n+1), fib(n) \rangle$

This now easily folds with (8) yielding the desired optimized function definition

(10) $fib_{tup}(n+1) = \langle u1 + u2, u1 \rangle$, where $\langle u1, u2 \rangle = fib_{tup}(n)$

Remarks

The following remarks may be made concerning NPL's deployment of the fold/unfold technique:

1. In general, the fold/unfold strategy eureka steps correspond to the introduction, by the user, of an auxiliary function which, by the use of abstraction, defines the target program in terms of the source. Clearly, a desirable goal of transformation systems is to circumvent the eureka step by providing

target definitions *automatically.*[5] As a general rule, the larger the class of programs which can be successfully transformed by a system then the more user interaction is required to provide the associated *eureka steps.*

2. Considerable user interaction is required to guide the fold/unfold process. The construction of an appropriate sequence of unfoldings, foldings, and re-writings is almost as difficult as the construction of a formal proof of a program.[6] That is, the search for a suitable *fold* presents control problems. So, recalling the previous remarks, the two most problematic steps in the unfold/fold strategy are:

   (i) the *eureka step*: obtaining the initial definition of the target in terms of the source; and

   (ii) the control problems associated with when to apply the *fold* re-writing step(s) which eliminate any reference to the source definition from the target recursive step.

   This control problem is compounded by the trade-off between the degree of automation and the size of the class of *fold/unfold* transformations one wishes the system to encompass (henceforth, the author will refer to this trade-off as the *automation trade-off*). Darlington's attempts at partially automating his fold/unfold technique were, by his own admission, blocked by the fact that the heuristics he used only covered a fairly small class of problem, and were not flexible enough to be used uniformly (Darlington, 1981a).

3. The following remarks, (a) - (e), re-iterate what we said in §3.1.1, only within the context of Darlington's transformations.

   (a) In NPL both the specification and target language are recursion equations. However, the language is not a suitably expressive (or declara-

---

[5](Chin, 1990) addresses this goal within the context of tupling transformations.

[6]Recall that these are *not* the same: a formal proof will incorporate a verification that the program computes the desired (specified) input/output relation.

tive) specification language: i.e., it is not easy to express what a program should compute without worrying about how it should be computed.

(b) The original fold/unfold strategy, as it was presented in (Burstall & Darlington, 1977b), was not provided with a *correctness guarantee* for the source to target transformations: the system itself performs no verification checking, the onus being on the user to accept or reject the "improvements" made at each stage of the transformation. However, later incarnations have been shown to have a partial correctness guarantee for specified classes of functions (notably (Darlington, 1989; Chin, 1990) and (Tamaki & Sato, 1984)) – see next remark.

(c) With systems based upon the NLP design, a correctness guarantee for the source to target transformations can, *in principle*, be provided without *directly* using any recursion/induction principle but by a sequence of identities, or equality lemmas, where each identity corresponds to one of the six rules (R1 - R6) of the transformation system. That the equality lemmas are indeed equivalence preserving must *necessarily* be proved if they are, collectively, to ensure correctness of any source to target transformation.

(d) Furthermore, each extension to the class of functions requires a corresponding extension to the set of identities, or equality lemmas, which in turn will require a corresponding extension to the (set of) equivalence proofs.

4. Darlington's system provides some useful methodological tips for optimization (regardless of whether we are concerned with the direct transformation of source code, or with optimization through proof transformation). Below we sketch the strategy involved with tupling transformations (in *Chapter 6* we provide a general strategy for fold/unfold transformations):

   - Define a tuple auxiliary function, f_{tup}, in terms of the step cases of one's known function definition f.

- Try to evaluate the step case, $f_{tup}(n+1)$, of this tuple function, perhaps by using unfolding, folding and forced folding or something equivalent, such that
  - The auxiliary function definition is cashed out in terms which *do not* have recourse to the original definition and so,
  - the recursion schema employed in the original definition is transformed into a more efficient one, e.g. as in the linearization of *course of values recursion* into *stepwise recursion.*

A Further Simple Example

We now provide a further simple example – using the *append* function – which illustrates how the majority of *fold/unfold* systems operate.

1. *Inefficient version of* append:
   (a) $append(nil, Y) \Longleftarrow Y,$
   (b) $append(x :: X, Y) \Longleftarrow x :: append(X, Y)$ {:: is infix for cons},
   (c) $g(X, Y, Z) \Longleftarrow append(append(X, Y), Z).$
2. *Instantiated left hand side of the base equation:* $g(nil, Y, Z).$
3. *Instantiated left hand side of recursive branch:* $g(x :: X, Y, Z).$

The system then performs the following evaluations;

- **Base Case**
  - X in (c) is instantiated to the empty list nil.
  - (a) is *unfolded* with (c) to obtain:
    (d) $g(nil, Y, Z) \Longleftarrow append(X, Y).$
- **Recursive Branches**
  - X in (c) is instantiated to $x :: X$ to yield
    (e) $g(x :: X, Y, Z).$
  - (e) is unfolded with (b) to yield:
    (f) $g(x :: X, Y, Z) \Longleftarrow append(append(x :: X, Y), Z).$
  - (f) is folded with (b) to yield:
    (g) $g(x :: X, Y, Z) \Longleftarrow x :: g(X, Y, Z).$

So the completed efficient version of *append* provided by the system is as follows:

$$
\begin{array}{lcl}
append(nil, Y) & \Leftarrow & Y; \\
append(x :: X, Y) & \Leftarrow & x :: append(X, Y); \\
g(nil, Y, Z) & \Leftarrow & append(Y, Z); \\
g(x :: X, Y, Z) & \Leftarrow & x :: g(X, Y, Z).
\end{array}
$$

General Strategy for Fold/Unfold

Different transformation systems employ different transformation rules although it soon becomes clear that most share a common core, even if the jargon varies. This usually consists in some combination/variant of the six rules employed by the NPL system. The main cause for divergence of these systems, from the NPL system, is generally due to the particular *logic*, or perhaps *formalism* is more appropriate, used for the specification and/or target languages. Below, in **fig.** 3-1, we have abstracted a general strategic plan for *program* transformation systems from those systems which, like NPL, employ some variant of the fold/unfold strategy. Note that the strategy requires user intervention at several key points (1, 2, 4, 5). In particular: the generation of target definitions in terms of the source, 2; and the application of a *fold* in order to introduce a recursion into the target definition, 5. Note also that the tupling technique is subsumed by the general plan.

1. User inputs some form of *program specification*, at the very least specifying the input/output relation, including any specially defined procedures. The specification may or may not be computable.

2. User specifies a number of suitably instantiated goal equations E (*eureka step*).

3. Each of E is *symbolically executed* – unfolded and/or evaluated – repeatedly using some computation rule. The *computation rule* may be, for example, that of a functional language or logic language (the later usually being the standard computation rule for Prolog or some idealized version thereof).

4. Laws, or *lemmas*, are optionally applied. Control usually passed to user or determined by user provided heuristics/rules.

5. Further symbolic execution – unfolding – followed by folding is performed/attempted until a recursion is obtained

6. The final equations are ordered for computational use.

figure. 3-1: A General Plan for Fold/Unfold

There are many existing program transformation systems that employ some equivalent variant of the *fold/unfold* technique. As illustrated above, the strategy requires an *eureka step* followed by unfolding and then folding in order to introduce recursion into the target program. So, in order to achieve a complete transformation the system must have some means of *controlling* the folding and unfolding. Exactly *how* this is achieved is what distinguishes many of the current *program* transformation systems. Other distinguishing features include the transformation *application*, the *language of transformation*, and the degree of automatability/user intervention required to attain a complete transformation. By *language of transformation* we mean the form of the *fold/unfold* technique which they employ. This depends on the form of the equations being (un)folded which, in turn, depends on the particular logic/formalism employed (e.g. functional or logic). In the following sections we provide a brief survey of some of the more notable extensions/variations on Darlington's prototypic fold/unfold model.

Extended NPL Functional Transformation with Automated *Eureka*

Darlington's *Functional Programming Environment*, FPE, supports the transformational development of HOPE$^+$ programs, and as such is tailor made for tupling transformations. The FPE operates as a *transformation processor* that applies user-generated transformation plans, or *scripts*, to programs.

The transformations achieve some degree of automation by, in effect, carrying around a large open-ended tuple, or more precisely a variadic function which simulates the action of tuples, whose length is tailored to whatever the particular function undergoing transformation requires. This tailoring is controlled by the system containing a large "lookup" table which, via a complex management module, provides information on how to tailor the tuple length to the function's requirements. In fact, the system must contain quite *function specific* knowledge in order to account for the substantial creativity required to formulate clauses, additional to the ones describing the problem's logic, on which the folding can be performed. This method is successful for a fairly small class of functions, although interaction is still required in order to guide the management module. As a con-

sequence of the *automation trade-off*, the more complex the recursive equations of the initial source program then the correspondingly more complex the heuristic set has to be in order to tailor the open-ended tuple structure. Thus much of the elegance of the original NPL system is lost.

Green has investigated the application of meta-level transformation tactics which are partially specified within a meta-language, and identified by specific pre- and post-conditions (Green, 1991). These tactics can then be used to control the automatic transformation of programs within the context of the fold/unfold strategy. The selection of tactics is guided by special property-oriented abstractions on programs. These abstractions, in effect, allow the system to decide, heuristically, which of the tactics whose pre-conditions are satisfied is the most promising. However, once the tactic application has terminated, considerable further work is required to actually attain the optimized target: rather than producing a target program, the tactic applications produce an object-level transformation sequence. This may then require substantial refinement in order to specialize it to the program undergoing improvement. Nevertheless, the use of a meta-language to guide the automatic construction of a target transformation sequence is a neat approach, and one which is similar to using a meta-language in which tactics can be expressed in order to automatically construct proof-plans.

Green's research is still very much in it's early stages, but holds considerable promise toward automating a large class of fold/unfold transformations: the goal-directed reasoning at the meta-level, which is responsible for the planning strategies, operates within a much smaller search space than the object-level transformation space.

ZAP: NPL Functional Transformation with "Metaprogram" Control

Feather's system, ZAP, employs the six main rules of Darlington's NPL system, R1 - R6, together with a meta-program to control the transformation of a "proto-program", the latter comprising the source recursive equations (Feather, 1979a). The meta-program is composed of various special purpose re-write rules which

reduce the transformation search space. These re-writes take the form of rules specifying which functions are to be used for unfolding and which may occur in the transformed equations. As such they are more like meta-level re-write rules which dictate the application of Darlington's NPL type re-write rules.

The user must intervene in order to:

- input a *complete* specification of the program;
- introduce the auxiliary *eureka tuple* as in the case of NPL;
- supply each instantiated left hand side of the developing target, together with,
- a high-level transformation plan - a meta-level program - for the desired transformed equation (this places a very big onus for the transformation on the user);
- activate "default generators" which employ user-supplied type information in order to, for example, automatically generate the instantiated left hand side; and
- prove the re-writing lemmas applied by the system.

Although all transformations take place within Darlington's NPL system and formalism, Feather's extension does not benefit from the specification/target language uniformity: the initial user-directed transformations convert the specifications into a more efficient form suitable for direct translation to programs in a conventional high level language. Clearly there is potentially a large translation overhead here.

Furthermore, Feather emphasizes that the meta-level programs should be viewed as advice to the transformation system which in no way effects the correctness of the final program produced. In this way Feather's meta-level programs bear a similarity to the OYSTER proof plans which encapsulate, within a formal language, the

patterns of reasoning employed by humans when synthesizing programs by theorem proving in mathematics. The proof-plans can then be used to heuristically guide the proof development.

The main weaknesses of Feather's system are that the meta-level descriptions are themselves rather weak (consisting primarily of pattern-oriented transformations), and there is no real strategy for using the tactics.[7]

Fold/Unfold Program Transformations with Automatic Tupling

Although many of the fold/unfold program transformation systems rely on procedures for generating new predicates/functions, and their definitions, on which the folding can be performed there has, to date, been limited success in automating the *eureka step*, although this is surely wherein most of the "intelligence" of the transformation lies.

Recently however, Chin, a student of Darlington's, has described several methods for automatic program transformation within the HOPE system (Chin, 1990). Although Chin documents an impressive range of automatic methods for program transformation, which are currently undergoing implementation, the most relevant to this thesis is his description of automatic tupling techniques. By an analysis of *symbolic dependency graphs*, based on (Pettorossi, 1984), Chin is able to describe an automatic procedure for finding a pair of *matching tuples* by the unfolding of selected calls to the source program, and then using matching as a means of testing for successful folding. The process is best described by example. We shall again use the Fibonacci function.

Fig. 1–3, *chapter 1*, is an example of a dependency graph for a function call $fib(5)$. A dependency graph, DG, is a representation of a particular function call's evaluation tree which shows the calling structure of the subsidiary recursive calls.

---

[7]These weaknesses provided the motivation behind Green's development of transformation tactics with formally specified pre- and post-conditions. Similarly, as we shall elaborate in Chapters 4 and 5, the OMTS transformation tactics are provided with pre and post-conditions.

A *symbolic* DG is based on function calls which are potentially infinite in size. The initial portion of the symbolic DG for *Fibonacci* is shown below in **fig.** 3–2. The

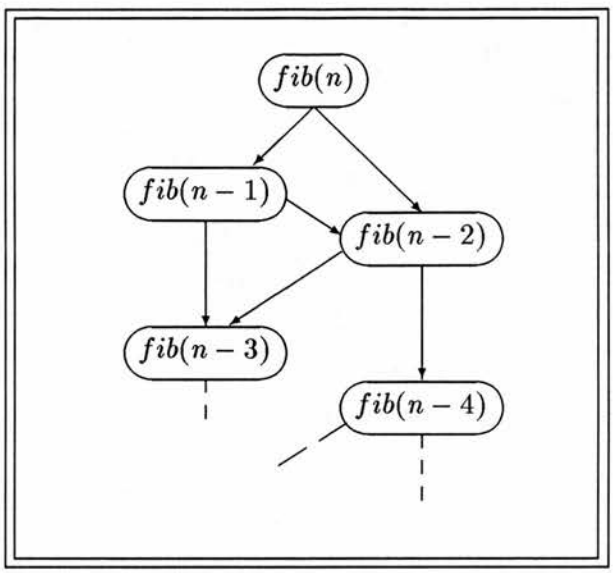


Figure 3–2: The symbolic DG for $fib(n)$

multiple evocations of subsidiary calls, the redundancy pattern, is exhibited by more than one arrow directed at any particular node.

The main idea taken from (Pettorossi, 1984) is that:

> An appropriate eureka tuple can be found if and only if there exists a *progressive sequence* of *cuts* that *match* one another, in the function's dependency graph.

A *cut* is defined as a subset of nodes across a dependency graph that when removed will divide the graph into two disconnected halfs. A *progressive sequence* of cuts is a sequence of cuts ordered according to size (i.e., according to the number of nodes in the subset). A pair of cuts *match* if a consistent substitution can be obtained when each function call of the first cut is matched with the corresponding function call of the second cut.[8]

The finding of an appropriate *eureka tuple* depends on the notion of a continuous sequence of cuts. This is defined by Chin as follows:

---

[8]These terms are *formally* defined in (Chin, 1990).

> A *continuous* sequence of cuts, $cut_1, cut_2, ..., cut_N$, is a successive series of cuts which starts with the root node as its first cut. This sequence successively obtains the next cut by giving up a subset of nodes... from the *topmost set* of the current cut in order to acquire the children for the next cut.

The topmost set of a cut is defined as a set of nodes whose ancestors are not present in the cut itself.

Returning to the example and starting with the main function call, Chin's analysis replaces $fib(n)$, the first cut, with its two subsidiary calls, $\langle fib(n-1), fib(n-2)\rangle$. This gives us the second cut. The analysis then proceeds by unfolding only that call in a cut which is *not* a subsidiary call of the other call, i.e., the topmost item. So, since the function call $fib(n-2)$ is a subsidiary call of $fib(n-1)$, only $fib(n-1)$ is unfolded. This gives the third cut, $\langle fib(n-2), fib(n-3)\rangle$. The third cut matches the second cut, thus providing the analysis with a matching tuple.

Chin's process is essentially the same as that described for Darlington's tupling technique: the fold/unfold steps required for the tupling transformation are achieved by locating a pair of matching tuples by the unfolding of appropriately selected calls and then using matching as a means of testing for successful folding. The main difference is that the use of such selection ordering allows for a considerable degree of automation, since once this analysis succeeds the main task of the tupling transformation – finding a successful fold – will have been achieved.

The automation hinges on the production of a continuous sequence of cuts. This in turn hinges on producing an appropriate *ordering* for selecting nodes to unfold during the analysis. So the production of (symbolic) DGs, and their subsequent analysis, are an essential ingredient of *automating* the tupling technique. The requisite dependency information simply is not present, for inspection and modification, in the (source) program code.

The use of dependency information for (automatic) tuple analysis, but within the context of proof transformation, is something we shall return to in *Chapter 5.*

In general, the tuple analysis is semi-decidable and non-deterministic. The analysis is not decidable since some programs will cause successive cuts to increase

in size. I.e., the analysis cannot handle programs with *dynamic tuples*, such as the binomial function:[9]

$$\begin{aligned} binomial(0, m) &= 1; \\ binomial(m, m) &= 1; \\ binomial(n, m) &= binomial(n-1, m-1) + binomial(n-1, m). \end{aligned}$$

The analysis is non-deterministic because there is often more than one way in which a selection ordering can be produced.

Finally, the reader should note the resemblance of Chin's matching tuple analysis to the process of *rippling-out* conducted during (OYSTER) inductive synthesis (*Chapter 2*, §2.2.4): in both cases successive unfolding is driven by the requirement to find a match with a known function definition. In *Chapter 5* we shall see how the rippling-out process can be exploited for similar purposes as with Chin 's matching tuple analysis.

Transformation of Functional Programs Using Fold/Unfold Technique *and* Adaptable Specification

Manna and Waldinger have developed a synthesis/transformation system, SYNSYS, which is almost identical to the NPL system except they make use of a special purpose specification language which facilitates the problem description, or specification (Manna & Waldinger, 1980). This means that the target language (LISP) is not the same as the specification language and there is an according increase in the complexity of the transformation operations. However, this specification language may be extended indefinitely by the user and adapted to suit particular situations: transformation rules are supplied for each construct in the specification language, to transform it eventually into a "primitive program".

The SYNSYS transformations are automatic. However, although there may be no user interaction during the transformation, there is good deal of user provided

---

[9] A program that uses a *dynamic tuple* is one that may require *different* sized tuples at each successive recursive call.

information – in the form of context-specific re-write rules – prior to the transformation: one of the main motivations behind SYNSYS is that one can *extend* and *adapt* the specification language to deal with particular function's requirements. In this way, by tailoring the specification language, the control problems associated with the fold/unfold methodology can, to some extent, be avoided. However, to be able to handle a large class of transformations, the SYNSYS system requires a large number of transformation rules, each of which, if source to target correctness is desired, require an equivalence proof.

Equivalence Preserving Fold/Unfold Transformations

Sato and Tamaki's logic transformation system, (Tamaki & Sato, 1984) is essentially the same as Darlington and Burstall's fold/unfold system, the main differences being:

- logic programs, as opposed to functional programs, are transformed (specifically *pure Prolog*); and
- emphasis is placed on the correctness of transformation which is not guaranteed by Darlington and Burstall's system.

Recalling §3.1.1, the general "advantage" of declarative programming languages is that the program need only specify the relation between input and output leaving the processor to deal with how output is computed from input. However, considerations concerning efficiency of computation remain unaccounted for. The motivation behind the transformations is to equip the declarative programmer with tools for attaining efficient declarative programs:

> To make the declarative programming style a real advantage of logic programming, we need a programming environment where lucid, specification-like programs are automatically or semi-automatically transformed into less lucid, efficiency-oriented programs, (Tamaki & Sato, 1984).

The second main difference is that Tamaki and Sato's system guarantees equivalence for *each* specific transformation rule they apply (i.e., a correctness proof is

provided for their system). This is not really a very important difference since, although Darlington and Burstall's original implementation of the fold/unfold strategy was not provided with a correctness guarantee, a correctness proof for their transformations can be provided by elaborating their functional formulation and by adding a formal semantics to the language. Indeed, this has since been done within Darlington's *Functional Programming Environment*, FPE, which supports the transformational development of HOPE$^+$ programs.

"Standard" Representation to *Difference-list* Transformation

Grant and Zhang have presented an algorithm for automatically transforming *Prolog* programs into their equivalent difference-list forms which exhibit more efficient list-processing behaviour (Grant & Zhang, 1988). If we denote a difference-list by L/E then this means *the difference between* L *and* E i.e., L/E represents the part of L with E removed. The reverse of a list using a representation whereby the first argument is represented as a difference list would be as follows:

$$reverse([], L/L),$$
$$reverse([H \mid T], L/R) : -reverse(T1, L/[H \mid R]).$$
$$rev(L, R) : -reverse(L, R/[]).$$

There are no calls to append in this definition resulting in this being an $O(n)$ algorithm as opposed to the naive version which is an $O(n^2)$.[10]

The approach Grant and Zhang take in transforming "standard" Prolog procedures into their corresponding difference-list representation shares much in common with that of Darlington and Burstall's fold/unfold technique: new definitions are supplied, the *eureka step*, to allow (un)folding with the original source definition equations and/or any other equations in the current equation set. Several heuristics are supplied for the control of the fold/unfold technique. Grant and

[10]I.e., the naive version makes of the order of n^2 recursive calls whereas the difference-list version makes of the order of n recursive calls.

Zhang also employ further techniques in their transformations such as partial evaluation, procedure combining and data structure mapping.

The *automation trade-off* clearly affects Grant and Zhang's system: the transformation deductions, in particular the rule set used to form new procedures and definitions, are in danger of leading to a combinatorial explosion. The heuristics employed to guide the fold/unfold technique are hence somewhat specific to the difference-list transformation domain. It is this limited application which allows for the degree of automation (including that of the *eureka step*) achieved by the system.

Transformation of Annotated Logic Programs

S. Gregory has investigated the feasibility of "compiling" logic programs bearing the control annotations of IC-Prolog into sequential logic programs (Gregory, 1980). This is achieved by the application of the classical *fold/unfold* technique. Indeed, Gregory stays close to Darlington's NPL transformation methodology, the main differences with his system being:

- the application (namely, the transformation of control annotations of IC-Prolog into sequential, and annotated, logic programs); and

- the use of Horn clauses for both the specification and target languages, thus allowing for simpler transformation rules (although this approach renders the specification language less powerful).

No correctness proof is provided, although we may assume that, since Gregory essentially employs the six main transformation rules of the NLP system, that the transformations are, at least in principle, equivalence preserving. Gregory also shows that the compilation of the resulting annotated logic programs can be partially automated.

Horn Clause Program Derivation from Standard Logic Specifications

C. J. Hogger regards program derivation as the top-down symbolic execution of a standard logic form specification, (Hogger, 1981; Hogger, 1980). The main technique of the system is, again, based on Darlington's *fold/unfold* approach, except here we are concerned with attempts to synthesize *Horn clause* programs as opposed to NPL type functional algorithms. Similar to the OYSTER synthesis philosophy, Hogger views program construction as a goal-oriented derivation. The characterizing properties of Hogger's system are:

- the *specification* comprising a set of *axioms* defining the desired input-output relation to be computed;

- A special purpose *logical deduction system* used to derive a set of computationally useful Horn clauses;

- the derivations, which are equivalence preserving, consist of sequences of rule applications which are classified as follows:

  Goal simplification: replaces the current goal by logical implication,

  Goal substitution: introduces new information by substituting (sub) terms in the current goal for (sub)terms in the specification axioms – Goal substitution is user-activated by a "call";

- a variant of the *eureka step* for lemma generation: recursions are introduced into the derived clauses by a tailored version of the *folding* rule (this recursion introduction process being very much user-guided).

- a termination point corresponding to (i) failure or (ii) a successful final goal constituting the body of the derived procedure, whose head is the current substitution instance of the initial goal.

The input to the system is a single "call" for which a procedure is sought. After the application of the above rules, the resulting derivation *roughly* resembles a conventional top-down logic program.[11]

Clark also treats program derivation as the top-down symbolic execution of a standard logic form specification (Clark, 1979; Clark & Darlington, 1980). He does, however, augment his system with further derivation rules which perform more sophisticated pattern matching, and subsequent substitution, operations by making greater use of terms in the derivation process.

An important motivation behind the systems of Hogger and Clark was the development of a suitably declarative specification language. However, more recent systems, such as Darlington's *Functional Programming Environment*, NuPRL and OYSTER are equipped with better facilities for supporting the specification of problems (as opposed to their procedural solutions).

3.2.2 Transformations Based On Explanation Based Learning/Partial Evaluation

The *explanation based learning* approaches to program transformation all have in common the use of either a *particular* instantiation of, or an abstract input to, a source program in order to *guide* the development of a *general* target program. By observing the behaviour of the source program, when run on a concrete or abstract query, the system, or user, can modify that behaviour according to the desired transformation application (hence the correlation with partial evaluation). The applications differs considerably. In (De Schreye & Bruynooghe, 1989) an example driven transformation is used to remove redundancies from simple function definitions by controlling the *folding* and *unfolding* of equations in the current equation set. In (Bruynooghe *et al*, 1989) a source program is improved for a small number of abstract inputs such that the target program improves on the

[11] This is because a call need not be atomic, and the replacement of a call by a "body" is determined by one of the substitution rules.

execution speed of a given Prolog program, by manipulation *only* of the computation rule under which it is executed. That is, the Prolog *compiler* is optimized. In (Huet & Lang, 1978) explanation based learning techniques are a suggested means of transforming past successful source to target transformation sequence. This involves forming a generalized program "prototype" from the source, again by executing the source program on a specific query.

The relation - or, arguably, equivalence - between partial evaluation and EBL has been studied in (van Harmelen & Bundy, 1988). Intuitively speaking, we can see how the two techniques merge within the optimization field since EBL guided transformation *just is* using a specific example of the source - a specific partial evaluation - to drive the fold-unfold transformation of the source such that repeated/identical subcomputations are removed.

Explanation Based Learning Transformation of Logic Programs

We shall henceforth refer to the transformation strategy of Brunynooghe, De Raedt and De Schreye of applying *explanation based learning* techniques to the *program* transformation domain as EBL transformation (De Schreye & Bruynooghe, 1989). There is, as of yet, no implementation, although Brunynooghe *et al.* provide a fairly detailed description of the transformation methodology.

In general, explanation based learning involves using a *specific* example to form a generalized "prototype" by substituting sub-terms for variables. Then by subsequently re-instantiating the "prototype" with target sub-terms we, hopefully, arrive at a solution sequence to the target problem.

Basically, Brunynooghe *et al.* suggest the removal of redundancies – repeated subcomputations – from programs by observing the behaviour of particular examples (i.e., the input is fixed).

Once again, the fold/unfold technique is used in the program transformation but by using a fixed input example to *control* the (un)folding, Brunynooghe *et al.* are able to automate (in principle) the crucial folding of subgoals in order to create new predicates. This automation is, however, also due in no small way to

the system being limited to a very specific class of transformations, namely those which result in the removal of identical sub-computations (again, a consequence of the *automation trade-off* between the degree of automatability and the size of the class of transformations).

Brunynooghe *et al.* suggest working with logic programs (specifically Prolog). The initial *source* program for the EBL transformation of *Fibonacci* will be equivalent to the *course of values* definition of *section 2.12.* As far as the fold/unfold technique is concerned nothing is *essentially* gained by this although the declarative nature of the language does allow for easier manipulation and we can see exactly what is going on in the transformation process.

If we look at the computational tree displayed in **Fig.** 1–3, *chapter 1*, we can see that in order to evaluate $fib(5)$ numerous repeated sub-computations are performed (for example, a call on $fib(2)$ appears 3 times in the tree). It is such *redundancy information* which guides the folding responsible for introducing new predicates.[12] So what the EBL transformation system does is to execute a query, in this case $fib(5)$, and then observe any duplication of subgoals in the computational tree. The repetition of $fib(3)$ is eliminated by adding its output, 2, as an extra output argument to the subgoal $fib(4)$. This is realised by a basic *fold/unfold* technique which employs the *eureka step* to allow for folding. By unfolding $fib(4)$ both occurrences of $fib(3)$ are obtained in one goal statement, and the undesired one is eliminated by factoring (full details are provided in (De Schreye & Bruynooghe, 1989).

Compiling Control transformation of Logic Programs using Partial Evaluation

Brunynooghe *et al.* have also partially implemented a transformation system (Bruynooghe *et al*, 1989). The system is different from that outlined in [Brun-

[12]The term *redundancy information* is my own and was originally coined to describe the process of *pruning* of branches from an OYSTERproof tree which result in redundant computation (*cf. chapter 5*).

ynooghe *et al.* 88] but also borrows from EBL techniques. EBL, effectively *partial evaluation*, is employed in a novel technique for avoiding the overhead caused by the execution of control languages when executing a program under the standard computation rule for *Prolog*. Hence, execution speed is increased by manipulation only of the computation rule under which a given Prolog program is executed and not by logical transformation like that between naïve and accumulating reverse.

This system is less versatile than the later system discussed above but is interesting since there is no cause for an *eureka step* during the transformation: the target program realizes a Prolog computation which is equivalent to a computation of the source program. However, no computation rule makes provision for lemma generation - i.e., we cannot provide clauses, additional to those describing the problem's logic, upon which folding can be performed in order to yield new predicates (and definitions), which in turn are used to introduce recursion into the target procedure. So, for example, this system can *not* improve, automatically or otherwise, on the course of values definition for Fibonacci.

The system operates by following two procedures:

1. the production of a *symbolic trace tree*;

2. the production of a new program specialized only to admit the efficient execution of the program, even under the standard computation rule (the new program is referred to, by Brunynooghe *et al.*, as a *meta-interpreter*).

The processes required to realize 2 above hold much in common with *specialization* whereby we, in effect, partially evaluate a program's symbolic trace tree. The difference is that *specialization* amounts to a logical transformation of the program in question *and*, of course, we operate on synthesis proofs.

Similar in spirit to (De Schreye & Bruynooghe, 1989), the production of the symbolic trace tree involves improving the program by successively running it on a small number of *abstract* queries and inspecting the dependency information revealed by the (partially evaluated) traces. Each successive execution reveals new

dependency information which causes a corresponding modification of – redundancy removal from – the preceding tree. Much responsibility rests with the user to make good query choices such that the eventual target abstraction covers all the data for which the program can succeed, and the resulting tree is a correct abstract representation for all possible successful executions of the program.

Translation of Clausal Specifications

W. F. Clocksin describes a technique for translating numerical algorithms, specified as clauses, into data-flow graphs. These graphs have the desirable property that common subexpressions are computed only once. The translation is not, strictly speaking, a program (nor proof) transformation but is of interest since the technique may be extended to provide a general program transformation technique: by fixing the input of an algorithm, and using *partial evaluation*, we obtain an example computational tree wherein repeated sub-computations can be observed. Such observations can then be used to control the development of the target algorithm. Indeed, this is precisely how Bruynooghe *et al.* arrive at their system, and is similar in spirit to the *specialization* of programs (§3.2.3 below).

3.2.3 Program Adaptation/Optimization Through Proof Transformation

We now turn our attention to systems which transform programs through transformations performed on synthesis proofs. As far as the author is aware, there is only one such working system: Goad's specialization system.

The EBL approaches to program transformation, §3.2.2, provide the closest analogue to specialization: in both cases a target program is sought by (i) observing the (sub)goal inter-dependencies within a partially instantiated, and (ii) proof *pruning* any redundant proof (sub)trees accordingly. The main difference is that the goal of specialization is *not* a generalized solution but a target optimized for the *specific* input values chosen by the user.

In addition to surveying Goad's work, we also provide a brief account of a suggested system design, (Pfenning, 1988), for optimizing programs through the application of meta-level transformation operators.

Program Specialization Through Proof Transformation

Although perhaps not as well-known within the program transformation community as the influential work of Darlington *et al.*, (Goad, 1980b) and (Goad, 1980a) offer, the author believes, an equally pioneering body of research: the first working system which (necessarily) requires the use of proof transformation to achieve source to target program transformation. Goad, building upon the more theoretical work of Kreisel, (Kreisel, 1975; Kreisel, 1977), demonstrates how proof transformations – specializations – can improve the efficiency of extracted programs in situations where a general purpose program is applied to inputs satisfying some given constraints.

The constraints are realized through partial evaluation on the input parameters of the program (dubbed *initialization*), and the transformations essentially consist in the pruning operations described in §2.3.3 (thus we do not provide an account of the specialization methodology in this chapter). They are:

> **Normalization**: which performs optimizations by the removal of any case split branch in the proof tree whose corresponding case condition evaluates to false (when evaluated by the initialization stage).
>
> **Dependency pruning**: which performs optimizations by the elimination of case analyses – *cut elimination* – whose outcome was decided by formulae already assumed on the branch so far taken in the proof tree.

The use of proof transformations is essential since the functionality (input/output) of the specialized program in general may be different from the functionality of the original program, but they both satisfy the same specification (where, as usual, program transformations do not have a specification present, and hence transformations have to be restricted to those that preserve input/output behaviour).

Goad notes, as we did in *Chapters 1 and 2*, that constructive proofs of program specifications differ from straightforward programs in that more information is formalized in the proof than in the program, and that therefore proofs lend themselves better to *transformation* than programs since "one expects that the data relevant to the transformation of algorithms will be different and more extensive than the data needed for simple execution", (p. 40 (Goad, 1980a)).

A sizable amount of the extra information in the more unnatural proofs is precisely information concerning how to *efficiently* compute the input/output behaviour synthesized in the natural, but inefficient, proof (§2.2.8)

• *Some Properties of Goad's System*

At the outset of (Goad, 1980a) the following points are made:

1. The partial evaluation can be done on an incomplete proof with unproved lemmas without compromising the computational usefulness of the proof as a whole.

2. Although specialization followed by pruning is not *guaranteed* to decrease the *execution time* of an algorithm, it will do so most of the time simply because its purpose is to tailor algorithms (or proofs) to a specific task, or rather to a specific class of input. Pruning is, however, guaranteed to reduce the *size* of the algorithm.

3. Pruning is guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. That is, given the constructive proof and a partial evaluation, pruning is guaranteed to prune *only* the computationally redundant parts of the proof tree without effecting the input/output behaviour *relative to the desired partial evaluation of the function parameters.*

4. Conventional computational descriptions (such as the conditional form or some logic programming description) are *not* subject to the pruning trans-

formations. This is because any valid transformations on conventional descriptions must preserve extensional meaning since they only contain information about the function to be computed. This is a nice illustration of the benefits of proof transformation as opposed to program transformation.

- *A Simple Example*

To illustrate specialization and pruning, Goad uses the following algorithm for computing an upper bound for both the sum and product of two positive rational numbers x and y:

- **(1)** $u(x, y) = x \leq 1$ then $(y + 1)$ else $(if\ y \leq 1$ then $(x + 1)$ else $2xy)$.

The algorithm specified above is in its *conditional* form and as such may only be slightly simplified as a result of partial evaluation. Suppose the value 0 is supplied for y, this results in the "specialized" conditional:

- **(2)** $u(x, 0) = x \leq 1$ then $(0 + 1)$ else $(if\ 0 \leq 1$ then $(x + 1)$ else $2x0)$,

which, upon evaluation, reduces to:

- **(3)** $u(x, 0) = x \leq 1$ then 1 else $(x + 1)$.

This simplification corresponds to the first stage of pruning: *normalization.*

The formalization of the upper bound algorithm as a *constructive existence proof*, its subsequent specialization (partial evaluation) and the application of normalization pruning then allows $u(x, 0)$ to be *automatically* simplified, by the use of dependency pruning, to the expression:

- **(4)** $(x + 1)$.

This is because the constructive existence proof will contain a case analysis whereby the case split is dependent on the size of x. Now, the fact that $(x + 1)$ is an upper bound for both $(x + 0)$ and $0 \times x$ does not depend on x being greater

than one. This dependency information is contained in the proof and, via partial evaluation and pruning, allows the removal of the "computationally redundant" case split according to the size of x. Note that **(3)** and **(4)** are different functions (e.g. different input/output behaviour is observed for $x = 0.5$). However, as far as the partial evaluation (specialization) is concerned, in this case y being set to 0, subsequent *normalization* will preserve input/output behaviour. In other words whilst normalization will transform the algorithm, reducing its size, it is guaranteed to preserve the input/output behaviour. Dependency pruning, on the other hand, may change *both* the algorithm and the function.

The representation of the conditional expressions, (1) - (4), as natural deduction proofs, based upon the Prawitz natural deduction system, allows for the pruning transformations to be performed automatically (Prawitz, 1965).[13] This is schematically represented in **fig.** 3–3, where $\stackrel{I}{\Longrightarrow}$ and $\stackrel{O}{\Longrightarrow}$ signify the input and output respectively.

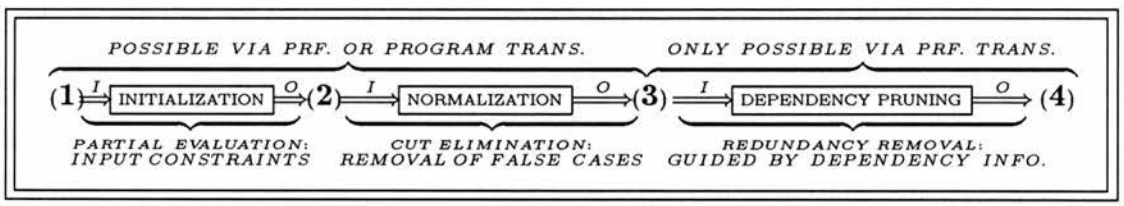


Figure 3-3: The Control Flow of Goad's Specialization Process

- *The* P-Calculus

In fact Goad does not directly use the Prawitz proofs, but rather performs the pruning transformations on *abstractions* from the proofs, *p-terms*, that are formulated within a special purpose calculus, the *p-calculus*. The structure of proofs

---

[13]We need not concern ourselves with the precise nature of the Prawitz natural deduction system since Goad infact employs a deduction system only loosely based on the Prawitz system (see next section). For the purposes of this thesis, the Prawitz natural deduction system can be viewed as similar to the sequent calculus, *Chapter 2* (especially §2.2.2), *without* actually employing the sequent syntax (i.e expressions of the form $A \vdash B$). There are, of course, other differences,some of which do have computational significance. For these the reader should consult (Prawitz, 1965).

is preserved in the extraction of *p-terms*. A *p-term* is, in effect, a skeleton of a proof in which the inference rules of the proof, but not the formulae to which they are applied, are recorded. Open assumptions in a proof are mapped to free variables in the extracted term, and discharges of assumptions correspond to bindings of variables.

The p-calculus is derived from the λ-calculus with pairing by the addition of the operators OE and EE, which correspond directly to the $\vee$-elimination and $\exists$-elimination inferences of natural deduction. The OE operator is interpreted as a *conditional* operator which keeps track of the dependency information needed for pruning. The procedure by which *p-terms* are extracted from proofs is based on the same correspondence between inference rules and functional terms that is used in constructing descriptions of functions in the λ-calculus (viz. the Curry-Howard isomorphism, *cf.* §2.2).

However, although an extract algorithm, a *p-term extract*, can be extracted from the *p-terms*, a constructive proof (Prawitz or otherwise) of the specification computed by the *p-term* extract cannot be obtained from the *p-term*. In other words, although the *p-terms* are akin to proof-plans in the sense that they represent skeleton proofs, they are not akin to proof-plans in the sense that, if applied to a program specification, they will produce the requisite proof (and thereby *proof-extract*. The "*p-term* methodology" is beneficial in terms of the speed of the transformation process: the *p-terms* contain just, and only just, the right amount of information required for the (automatic) pruning transformations *and* for the (automatic) extraction of a (target) algorithm. This minimizes the amount of information that is subject to manipulation in the course of transformation.

However, because the *p-terms* lack some essential ingredients as far as proofs are concerned – notably any verification component – then any correctness guarantee of the pruned target with respect to a target specification is lost: simply because there is no target specification nor corresponding proof. It is therefore fortunate that the dependency pruning transformations are guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. For, as we have seen, they are not equivalence pre-

serving, and Goad's system has no target specification, and corresponding proof, with which to prove the correctness of the specialization process.

So, somewhat surprisingly, Goad does not exploit the properties of existence proofs to obtain a correctness guarantee for his transformations, but rather, due to the fact that the target program is not extracted from a *proof* of the target specification, relies on the method common to most program transformation systems of using correctness preserving re-writes. Hence Goad's system inherits the drawback that any extensions to the transformation rules, pruning or otherwise, will require (if possible) proofs that they preserve equivalence.

Furthermore, we may desire modifications on a source program's functionality which are intentionally designed to modify the source specification. In such a case, if the target program is extracted from a *p-term* then there is no proof of a target specification (i.e., modified source specification) with which to judge the correctness of the target program. An example of such a specialization, *induction grounding*, was described in §2.3.3. In *Chapter 4* we describe in detail how the OMTS performs induction grounding (in addition to normalization and dependency pruning).

To avoid repetition, we shall reserve what more we have to say regarding the correctness of the specialization transformations until *Chapter 4*, where we compare Goad's methodology with the author's rational reconstruction within the OYSTER environment.

It should be noted, however, that the lack of any target correctness check (w.r.t any target specification) is more or less in keeping with Goad's motivations, which are more concerned with *computational usefulness* and *not* correctness (*cf.* point 1 of the itemized properties of Goad's system). Such considerations of the speed of transformation, presumably, account for the sacrifice of a proved specification for the target in favour of the austerity of information that undergoes manipulation.

A Methodology For Program Through Proof Transformation Using Meta-Level Control

Pfenning has recently sketched a methodology for program transformation through proof transformation by using LF, (Harper *et al*, 1987), to formulate meta-theorems which can then be used as tactics for transforming object-level proofs [Pfenning 89]. This differs from our approach of using Prolog to specify the tactics. However, although no implementation is yet available, Pfenning is in agreement with us that NUPRL type proof development systems would be particularly suited for providing the object-level proof structures.

Similar to the approach of the author's OMTS system, Pfenning suggests using higher order Martin-Löf type theory as a medium for program optimization. To quote from (Pfenning, 1988):

> We present a methodology for deriving verified programs that combines theorem proving and transformation steps. It extends the paradigm employed in systems like NUPRL where a program is developed and verified through the proof of the specification in a constructive type theory.

The author has been in personal communication with Pfenning, primarily to determine whether or not his high-level design accords with the author's OYSTER implementation. Pfenning intends to operate within the same class of transformations as those performed by the OMTS: the introduction of a lemma followed by a number of *proof reductions.* Pfenning also regards the correctness guarantee of the source to target transformations, that is bought by virtue of the presence of a (complete) target specification, as a benefit of the proof transformation approach. Furthermore Pfenning observes that the meta-programs, or tactics, that apply the proof reductions can themselves be extracted from proofs of theorems which guarantee that the transformations are equivalence preserving.

Pfenning suggests using LF as the formal system for describing a logic. The operational interpretation of the tactics would be based on ideas from λ-Prolog

(Miller & Nadathur, 1988).[14] The reason given for the choice of LF as the meta-logic, is that it has the expressive power to describe a wide class of transformations. This allows for the formal statement, and proof, of *meta-theorems.* The proof of a meta-theorem then may be used to transform proofs in the object logic.[15] Pfenning holds that LF is better suited as a meta-logic than NUPRL since it is better equipped for deriving composite transformations from the definition of tactics.

A disadvantage of having distinct object and meta languages is that this can lead to a proliferation of complexity. For example, we cannot use the same unification algorithm for both inference and control reasoning.

Clearly, Pfenning's research has relevance to the author's, although, unlike the work of Goad, it did not provide much motivation for the bulk of the research in this thesis (which was near completion when the author learned of Pfenning's interests). It will, however, prove interesting in the future to see how Pfenning's approach of using two distinct languages, NUPRL for the object-language and LF for the meta-language, compares in practice to the approach taken with the OMTS. A foreseeable advantage of using an amalgamated logic, like LF, is that one avoids the proliferation of complexity that can occur with distinct object and meta-languages. For example, we cannot use the same unification algorithm for both inference and control reasoning. A disadvantage is that function/predicate definitions in an amalgamated logic may, if care is not taken, violate consistency (for example, by having truth predicates we may run foul of Russell's paradox).

---

[14] λ-Prolog is a *higher-order* implementation of Prolog, such that it allows functions, or predicates, to be bound to variables, passed as parameters, and returned from function calls.

[15] This bears similarities with some of the suggested *extensions* to the OYSTER transformation system presented in *Chapter 6.*

3.3 Summary

We began by discussing the fold/unfold technique for program transformation within the context of Darlington's NPL transformation system.

We abstracted a general program transformation strategy from the fold/unfold systems reviewed. The *eureka step* and the introduction of a *fold* during the target development were singled out as the main obstacles to automation: user guidance was required for lemma introduction, or the eureka step construction, and for guiding the sequence of unfoldings which follow in order to find a fold (and thereby introduce the new target recursion schema).

We made the observation that one factor that compounds the control problem is a result of the nature of many of the recursive equation re-writing systems that employ the fold/unfold technique: such systems (or usually the system's user) have to direct the unfolding towards finding a fold in order to attain the desired recursion schema.

A further problem which compounds the difficulty of automation is the extent of the class of transformations one wishes a system to encompass. This generally increases the burden on the human user in guiding lemma introduction and controlling unfolding. We called this the *automation tradeoff.*

We noted that, of those systems that ensure the correctness of their transformations, many employ equivalence preserving equality (identity) lemmas, within some suitable logic sub-set, thus avoiding any direct need for lengthy verification proofs. They do, however, require individual proofs for each lemma, *and* for each extension to the set of re-writes employed, further lemmas, with corresponding proofs, are required.

We paid particular attention to Darlington's use of *tupling* for removing redundancy by grouping together a collection of potentially re-usable function calls. We also discussed Chin's extensions to the tupling technique, in particular with

regards to automation (Chin, 1990). The important features of automatic tupling were the use of dependency graphs in analysing repeated sub-computations.

We reviewed some of the influential program transformation systems in the literature. The majority of these employ some variant of the fold/unfold technique. However they also introduced new features such as some form of meta-level control, and the use of EBL techniques. Regarding the latter, partial evaluation is used to fix the input of a program. Repeated sub-computations are then removed from the *general* case by observing the behaviour of the partially evaluated particular example.

We provided a description and discussion of Goad's *specialization* system which adapts algorithms to particular situations through partial evaluation and pruning transformations performed on proofs. This enabled us to identify that part of the process which, if automation is a desirable goal, requires more information than the target program language provides. The missing information is *redundancy information.* Such information can be abstracted from the proof in the form of the dependency information between various proof (sub)goals and hypotheses (which is tantamount to the dependencies between facts involved in the computation of the proof extract program). We also made the observation that proof transformation provides a unique opportunity for transforming the functionality of a program whilst retaining the same specification. We noted, however, that, due to the system design, Goad does not exploit the nature of the proof specification language in order to verify the source to target transformations.

Apart from Goad, none of the existing systems approach program transformation through proof transformation. [Pfenning 89] has recently discussed how programs can be transformed by applying meta-level tactics to program synthesis proofs. This general idea has, prior to [Pfenning 89], been implemented by the author in the OYSTER proof development system.

Our discussion of Goad's system, and of Pfenning's system design outline, reinforced the advantages of the proof transformation approach to program specialization/optimization that we identified in §2.2.11. These are:

Regarding Goad: Existence proofs contain extra information which is extraneous to that required for simple execution, notably, dependency information and a verification of the method. A knowledge of the dependency information can then be used to guide the transformation process through the transformation search space.

Regarding Pfenning: The proof of the specification affords us with a correctness guarantee for all terminating transformations.

Finally, we noted, looking back to *Chapter 2* and forward to *Chapter 5*, that the fact that the majority of inductive synthesis proofs follow a "prototypical" strategy could be exploited, for purposes of generality and automatability, in the transformation of proof instances of such a strategy.

Chapter 4

Specialization: Program Adaptation Through the Partial Evaluation and Pruning of Proofs

4.1 Introduction

This chapter contains:

1. The main motivations behind reconstructing C.A. Goad's specialization process within the OYSTER constructive proof framework.

2. A description of the author's proof specialization system, implemented within OYSTER and consisting of three main operations – or *transformation tactics* – for transforming a source synthesis proof:

   (a) *Initialization* which partially evaluates a source proof by instantiating *some* parameter in the specification;

   (b) *Normalization* which is designed to remove those branches from the *initialized* proof tree which will always evaluate to *false*; and

   (c) *Dependency Pruning* which is designed to remove those branches from the initialized *and* normalized proof tree which result in *non-repeated* redundant computation (i.e., (sub)-computations that are redundant for reasons other than duplicity).

Note that both normalization and dependency pruning qualify as proof tree pruning transformations.

We also describe a third pruning mechanism which we have implemented, and which serves as an extension to the functions of Goad's specialization system (Goad, 1980b; Goad, 1980a):

(d) *Induction Grounding*, designed to specialize (adapt) recursive behaviour through pruning source sub-proof trees associated with the application of mathematical induction.

3. A discussion of the original aspects of the author's proof specialization system. This is achieved through a discussion of the main differences between, and improvements over, Goad's approach to specialization (and pruning) and that of the author's specialization system, the PSS, constructed within the OYSTER environment.

4. Examples of the PSS at work. In particular, we shall discuss the methodology and effects of:

   - the partial evaluation of induction schemata; and
   - pruning redundant, but non-identical (or non-repeated), computation (i.e., dependency pruning).

5. Concluding discussions concerning:

   - the automatability and correctness of the author's specialization transformations;
   - the reasons for specializing programs through proof transformations (as opposed to transformations performed directly on program code);
   - the main properties of the author's specialization system; and
   - a comparison of the properties of the author's specialization system with those of some of the transformation systems reviewed in *chapter 3*, in particular systems which incorporate *explanation based learning* and/or *partial evaluation* techniques.

6. Finally, we provide an overall summary of *Chapter 4*.

4.1.1 A Rational Reconstruction

The specialization system is a *rational reconstruction* of C.A Goad's specialization system in that it is only the high-level rationale, or methodology, which has been re-constructed within the OYSTER environment (i.e., the three stages: partial evaluation, normalization, and dependency pruning).

Specialization within the OYSTER proof framework is treated as a particular kind of program optimization through proof transformation: a program's behaviour is optimized for a particular type of input through proof tree *pruning transformations* performed on its synthesis proof.

The PSS satisfies the three main desirable criteria for a transformation system:

- *Correctness*: all transformed programs are correct with respect to their specifications.

- *Automatability*: the source to target transformation requires no user guidance.

- *Generality*: for any algorithm that exhibits the kind of redundancy referred to in 2(b) and 2(c), of §1, *pruning* is guaranteed to reduce the size of that algorithm. That is, pruning will decrease the amount of branching in the (symbolic) dependency graph, DG, associated with the algorithm through a corresponding pruning of the branching in the synthesis proof.

Both systems perform specialization through the partial evaluation of a proof, initialization, followed by the two pruning operations, normalization, and dependency pruning (2(a)-2(c) of §1).

An important feature of both Goad's specialization system and the author's reconstruction is that pruning improves the efficiency of a computation by changing the function which it computes.

The main advantage, and novelty, of the reconstruction is that although such pruning transformations adapt the *functionality* of the source algorithm, the (initialized) source to target transformations are, unlike Goad's system, correctness guaranteed (providing the algorithm is completely specified in the root node of the source proof).[1] We shall discuss this original property of the PSS shortly in *§4.2.2.*

In addition to 2(a)-2(c) of §1 above, we also describe the induction grounding pruning mechanisms, 2(d). These perform a particular kind of pruning transformation for *inductive* proofs which is designed to prune away all the unnecessary evaluation associated with the unravelling of a *partially evaluated proof induction schema.* By this we mean a sub-proof of a source synthesis proof in which some parameter has been assigned a value – a specialization – and where that sub-proof consists of those branches associated with the application of induction. In effect, induction grounding exploits the induction-recursion duality for the adaptation, via partial evaluation and proof pruning, of recursive program extracts to particular situations. Partial evaluation of proof induction schemata is also guaranteed to reduce the size of an algorithm. This is done by removing the computation associated with evaluating the recursion schemata.

4.1.2 Motivations and Intentions

In this section we expand on the motivating factors, given in *Chapter 1*, behind the author's research concerning specialization. These overlap somewhat with the motivations behind the broader based recursive program optimization system described in *chapter 5.*[2]

---

[1]By source proof here we mean the initialized proof, 2(a) of §1, to which the pruning transformations are applied, 2(b)-2(c) of §1.

[2]Many of the properties of both the OYSTER proof transformation systems will be covered in this chapter and need not, therefore, be repeated in *Chapter 5*, (cf. §5).

Reconstructing, in OYSTER, *a Proof Transformation System*: As stated in *Chapter 1*, apart from the author, and more recently Pfenning, Goad is one of the few researchers who approaches program transformation through the transformation of proofs, and, as far as the author is aware, Goad is the only other researcher who has implemented such a system.[3]

Goad's specialization process provides a concrete example of how the extra information formalized in proofs - *information which is not concerned with simple execution* - can be exploited in the *adaptation* of programs to special situations: conventional computational descriptions (such as functional recursive equations or some logic programming description) are *not* subject to the dependency pruning transformations. This is because any valid transformations on conventional descriptions must preserve extensional meaning since such descriptions only contain information about the function to be computed.

By reconstructing the specialization process within the OYSTER environment we see that the exploitation of this extra information is not limited to the loose variant of the Prawitz proof environment upon which Goad's system operates (Prawitz, 1965). In particular, we see that the OYSTER proofs contain all the dependency information required for the specialization transformations.

Developing Proof Transformations that Satisfy Desirable Criteria: The correctness, generality, and automatability criteria apply to the proof transformations performed by both the author's systems. In both cases this is largely due to the properties of the theorem proving and the OYSTER proofs themselves. That a *proof* transformation system could be constructed satisfying *all* three of these criteria was, in itself, a major motivation.

---

[3]Pfenning discusses the design of a proof transformation system, *cf.* §3.2.3, although no system has yet been implemented (Pfenning, 1988).

Regarding correctness, the PSS proof pruning transformations provide the first steps towards a system for transforming the *functionality* of a source algorithm, whilst retaining the correctness of the target *with respect to a target specification* (as opposed to providing separate correctness proofs for the transformation re-write rules). Furthermore, with certain transformations the source and target programs satisfy the same *full* specification despite the fact that the functionality of the source is modified (specialized).[4] This, we believe, is a novel property of transformation systems, and one which marks an advance upon Goad's system (we elaborate on this difference in §2.2.2).

A Feasibility Study: A related motivation for constructing the OYSTER *proof specialization system* is that it provides a good foundation for investigating further applications of program transformation through proof transformation (such as the optimization of recursive programs – the subject matter of *Chapter 6*).

Induction Grounding Provides a Practical Example: The PSS induction grounding transformations constitute the author's preliminary investigations regarding transforming recursive behaviour through transforming the dual induction, and as such offer a more practical application of specialization than some of the rather esoteric examples designed specifically to illustrate dependency pruning. However, unlike the optimization of recursive programs, discussed in *chapter 6*, induction grounding specializes, or modifies, a recursive program by *removing* the induction schema, and thereby the recursion schema, and replacing it with a *finite proof-rule*.[5]

---

[4]I.e., the system exhibits *source-to-target correctness*, as defined in §2.2.8, for certain transformations on a source programs functionality as well as exhibiting source-to-target correctness for all optimization transformations (the latter being the subject matter of *Chapter 6*).

[5]To prove, by induction, that a property holds for all natural numbers implicitly requires proving that it holds for each and every of the infinite natural numbers. Hence

Proof Transformations Reduce Synthesis Workload: Human theorem provers are usually trained to find short, elegant proofs rather than long opaque ones, despite the fact that the latter may yield more efficient programs. The PSS transformations, and in particular induction grounding, illustrate how proof transformations allow the human theorem prover to produce an elegant *source* proof, without clouding the design process with efficiency issues, and then to transform this into an opaque proof that yields an efficient specialized program.

Reacting to Specifications: The specialization of a program through the partial evaluation of the initial specification, and corresponding synthesis proof, provides a good example of how proof transformation can assist in modifying a program's internal structure in accordance with an initial modification of its specification.

A System for Pruning Non-repeated *Redundant Computation:* The majority of program transformation systems which optimize a source program by removing redundant computation that is, in some sense, exhibited by observing the behaviour of the source program when partially evaluated, are usually limited to the removal of *identical*, or *repeated*, sub-computations (e.g., (De Schreye & Bruynooghe, 1989; Hogger, 1981; Clark, 1979)). Once a program, or proof, has been partially evaluated this is a fairly trivial task. So the removal of *non-repeated redundant* computation, through dependency pruning, is a more interesting application (related to the fact that we thereby transform the functionality of the program), and is one which *cannot* be performed through the partial evaluation of programs.

---

induction can be correlated with an infinite proof rule. Conversely, to prove that a property holds for a finite number of objects requires only a finite proof rule (recalling §2.3.3).

4.1.3 Originality: Correctness Preserving Transformations, Induction Grounding, and Extendability

Before describing, by example, the PSS transformations, we shall elaborate on the main differences in design between the PSS and Goads specialization system. As well as illustrating the original aspects of the PSS, this will also provide, from the outset, a clear model of the PSS design.

C.A. Goad makes the argument that the two main types of methods for the execution of (synthesis) proofs are inadequate for the purposes of specialization (and, indeed, any transformations which require the exploitation of dependency information). These are:

> *Method 1: methods which operate by transformation of proofs themselves* (e.g., Gentzen's cut-elimination procedure, (Szabo, 1969), and Prawitz's normalization procedure, (Prawitz, 1965)).
>
> *Method 2: methods which involve the extraction of code* from *the proof* (e.g., the realizability interpretations of Kleene for arithmetic, (Kleene, 1945), and Kreisel for analysis, (Kreisel, 1959), and the Dialectica interpretation of Gödel, (Gödel, 1931)).

This is because, firstly, the normalization methods are unsatisfactory because of their inefficiency. The proof will contain large amounts of information which is irrelevant to *both* execution and pruning. This additional information is subject to extensive manipulation in the course of normalization. Secondly, and conversely, the extraction methods cited by Goad involve abstracting away all the additional information, specifically the dependency information, which is needed for pruning.

Hence, recalling §3.2.3, Goad's system involves the use of a special calculus, the *p-calculus*, which is:

> designed to provide expression for just that information contained in natural deduction proofs which is needed for execution and for the pruning operations. (Goad, 1980b), p47.

As such, however, the p-calculus structures, or *p-terms*, do *not* contain all the information required faithfully to reproduce the *complete* proof from which it is abstracted nor do they contain enough information to construct (automatically) a target proof following transformation. In particular, the source and target p-terms contain absolutely no record of the verification components of the respective proofs.

So since it is the p-terms that are transformed, and then executed (through a p-calculus program extraction process), then, at the termination point of Goad's specialization process, there is no target proof, nor target specification. Nor is there any normalized proof, or corresponding specification, output by the intermediate normalization pruning operations. Hence, regarding Goad's transformation of p-terms, we have no correctness guarantee in either of the following two senses (recalling the distinctions drawn in §2.8):

(i) Regarding *program correctness*: we cannot check that the dependency pruned target program is correct with respect to a (complete) specification appearing in the root node of any target proof.

(ii) Regarding *source-to-target correctness*: we also cannot check that the target program is correct with respect to any proof specifications embodied in the root nodes of the initialized, and subsequently normalized, proofs. So, following dependency pruning, there is no *direct* guarantee that the program will compute the desired specialized input/output relation.

Recalling *chapter 3*, Goad's pruning transformations are, however, guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. So whereas there is no direct procedure for checking the correctness of the target – since there is no target specification – Goad relies on the properties of the transformations, rather than the proofs, to ensure correctness.

Goad's approach has the drawbacks that, firstly, any extension to the set of transformations (pruning or otherwise) must be coupled with additional proofs

that the extended set preserves the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. Secondly, for any transformations (pruning or otherwise) that are *not* designed to preserve source-to-target correctness (such as *induction grounding*), there is no means of establishing that the target is correct with respect to a target specification.[6]

Regarding the PSS approach, the provision of a correctness guarantee for transformations that alter a source programs functionality is a property of the transformation system, inherited from the properties of the object-level proofs. This is a novel state of affairs.

In **fig.** 4–1 we depict the following differences between the design of Goad's specialization process and that of the PSS.

- **At the termination of the PSS transformation we are provided with a target *proof* from which the target program is extracted** (abbreviated to λ–*ext* in **fig.** 4–1). This means that the target program is guaranteed correct with respect to the specification, S_{pe}, obtained by the initialization process of partially evaluating the source specification, S. The term *rule tree* refers to the PSS counter part to Goad's p-terms. The main difference being that rule trees do contain just the right amount of information required either faithfully to reproduce the source proof from which they are abstracted, or, after being subjected to the pruning transformations, to produce the complete target proof when applied to the source proof specification (more detail concerning the rule-trees is provided throughout §4.3). Hence the PSS normalization and dependency pruning transformations on the initialized source are correctness guaranteed in both senses (i) and (ii).

---

[6]We should again point out, as we did in §3.2.3, that Goad's motivations are more concerned with *computational usefulness* than with correctness. Indeed, even regarding the source program, Goad states from the outset that specialization can be done on an incomplete source proof with unproved lemmas without compromising the computational usefulness of the proof as a whole. He does not, however, say that this *does* effect the status of the objects of transformation as synthesis *proofs*.

- **The PSS has an additional transformation application: induction grounding.** We wish to make clear at the outset that although induction grounding *is* a form of pruning, it is neither equivalent to normalization, since no proof branches that yield false sub-computations are pruned, nor dependency pruning, since the pruning is not governed by the same kind of redundancy information, abstracted from the proof, or by the same pre-conditions that apply to dependency pruning (cf. §4.3). Induction grounding consequently has a wider field of application than the other pruning transformations.

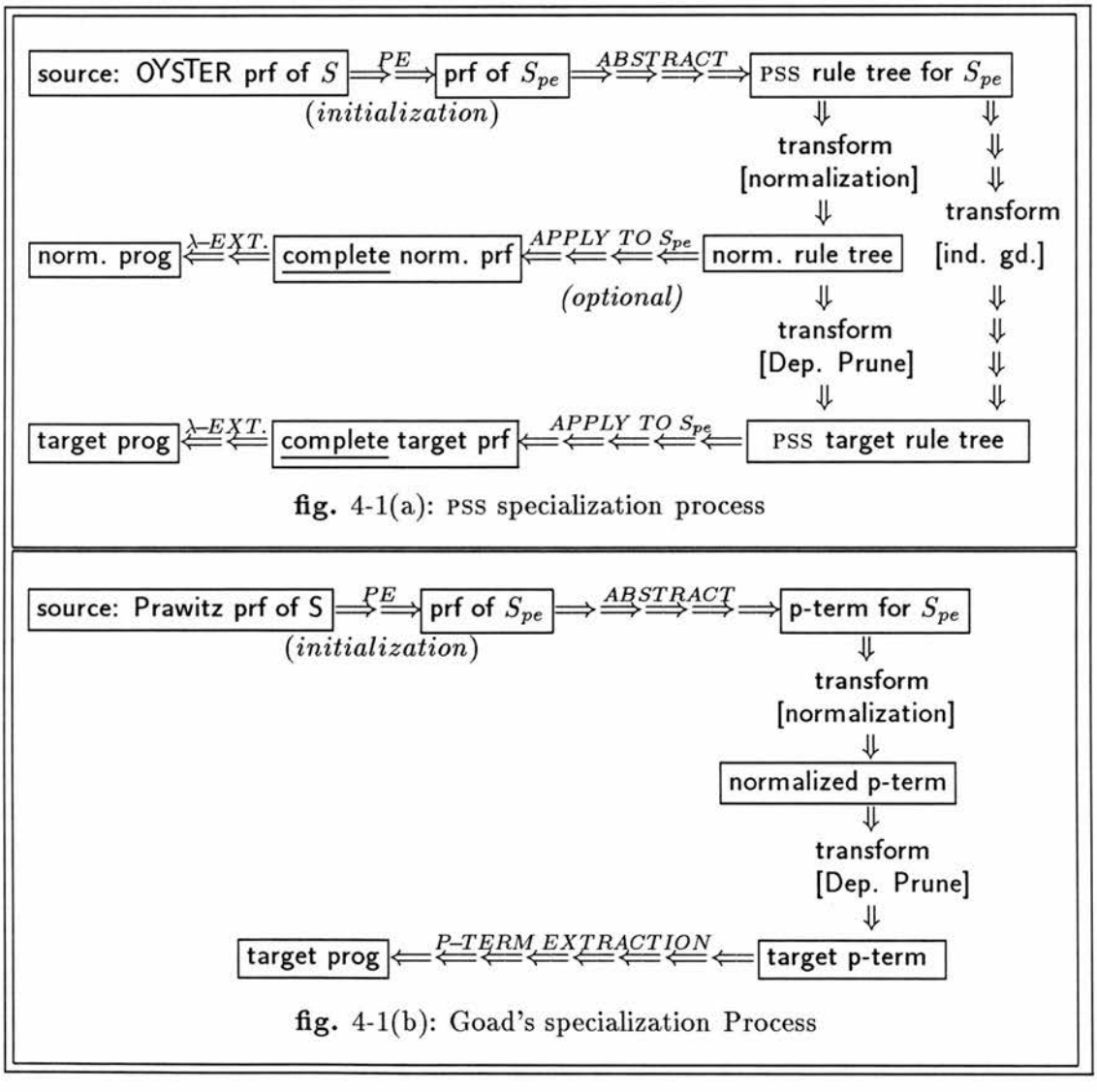


fig. 4-1(a): PSS specialization process

fig. 4-1(b): Goad's specialization Process

Figure 4-1: Comparison of Specialization Designs

- **The PSS is an "open ended" system.** The PSS provides scope for extending the specialization system such that the specialized target proof can be treated as an intermediary state of a broader proof transformation application.[7] Goad's system does not share this property because the target program does not have a corresponding proof, rendering any further program *through* proof transformations on the target impossible.

It should be emphasized that, whereas **fig.** 4-1(a) adequately depicts the differences between the author's reconstructed specialization process and Goad's original design (**fig.** 4-1(b)), the diagram may give the impression that the PSS specialization process is more rigid than in actuality (this being a general disadvantage of flow diagrams). In actual fact, each of the four PSS specialization transformations – partial evaluation (or initialization), normalization, dependency pruning and induction grounding – should be treated as distinct meta-level transformation tactics, with pre- and post-conditions (§2.3) that act upon the rule-tree proof abstractions. Since the rule-trees are, unlike Goad's p-terms, akin to proof tactics, or proof-plans, then each of the four specialization transformations are akin to tactic, or proof plan, transformers which can be applied independently to a proof rule-tree, and which will succeed in attaining the desired post-condition(s) as long as the pre-conditions are met prior to application.

4.2 The OYSTER Specialization System: the Specialization of Proofs in *Constructive* Type Theory

Corresponding to each example covered in this chapter, we provide below a brief explanation of what properties of the system the example is designed to illustrate.

---

[7]For example, whilst optimizing a source programs recursive behaviour we may (also) wish to specialize one of it's parameters.

4.2.1 The Purpose of the Examples

1. The *sumlist* example (§4.2.2): This example is primarily to explain:

   - the effects of specialization on the *length* of the input *list*, I.e., the partial evaluation of induction on the length of a list;
   - induction grounding, the effects of *pruning induction schemata* and the subsequent effect on the recursive behaviour of the extract algorithm.

   We also take the opportunity to elaborate on the function of the rule-tree abstractions, thus setting the scene for subsequent proof transformation examples (including those covered in *Chapter 5*).

   We note here, and elaborate in §4.2.2, that although this particular example involves the induction grounding of a proof by list induction, that the PSS is equipped to perform induction grounding on induction over other recursive types.[8]

2. The *sorting* algorithm example (§4.2.3): We give a brief example of the specialization of a more complex inductive proof (sorting by insertion, or *insertion sort*). The increase in complexity is marked by the source proof containing the following:

   - nested sub-proofs; and
   - nested induction schemata.

   This example illustrates how the system is capable of adapting algorithms containing *nested* recursion schemata through the pruning of *nested* induction schemata.

   We also take the opportunity, with this example, to illustrate the heuristic transformation setting of the PSS.

---

[8]Recall, from *Chapter 2*, §2.3.3, that we used stepwise induction over the naturals (*pnat* – *ind*) to illustrate, in the abstract, the effects of induction grounding.

3. The *upper bound algorithm* example (§4.2.4): The third example is a reconstruction of Goad's main example, the specialization *and* pruning of the *upper bound* algorithm.

   This example highlights the following features of the OYSTER specialization *and* pruning mechanisms:

   - *the specialization on the input argument values* – the partial evaluation of the types corresponding to the function arguments;
   - *both stages of the pruning mechanisms*: the normalization and dependency pruning of *constructive* proofs; and
   - the *correctness* of the pruning transformations.

The *upper bound* example clearly illustrates how a (target) program, that computes a different function to the source program, can be used to provide the *same* output as the source for a particular assignment to (instantiation of) one of it's parameters.

4.2.2 Example 1: Automatically specializing the *sumlist* OYSTER proof

The specialization of the *sumlist* proof corresponds to adapting a program which computes the sum of the integers in an input list of *any* length, to a target program which computes the sum for an input list of *specific* length, n. In conventional formalism, the source program is as follows:

$$sumlist([]) \Rightarrow 0;$$

$$sumlist([hd.tl]) \Rightarrow sumlist(tl) + hd.$$

This is represented within OYSTER as the following lambda calculus extract algorithm, where *list_ind* signifies an application of *list* induction:

$$\lambda l, list\_ind(l, 0, [v0, v2, v2 + v0])).$$

This states that the recursion is done on the input list l where the base case value for the empty list is 0, the head of the input list at each recursive pass is $v0$, the sumlist value for the recursive tail of the input is $v2$ and the sumlist value at each recursive pass is $v2 + v0$.

We can explain the source synthesis of the *sumlist* program by reference to the rule-tree abstracted from the completed source proof (displayed, together with an explanation, in **fig.** 4–2, p.163). We shall first, however, briefly recap on the function of the rule-tree abstractions.

Recall, from *Chapter 2*, that to avoid computational effort being expended on attempting to access individual semantic units the OYSTER representation of the source proof tree is processed, by abstraction, into a more accessible list structure, the rule tree.

The source proof is transformed by the application of transformation tactics to the source rule-tree. Using the transformation tactics, (sub)branches of the source sumlist proof can be accessed and the appropriates transformations made.

The rule tree contains:

- the branching structure of the proof;
- the rules applied along with any corresponding arguments; and
- an account of the dependencies between facts involved in the computation.

The dependency information has to be abstracted from the rule-tree, which retains a record of the inter-relations between *proof hypotheses* and sub-goals.

At the termination of a transformation the system will have constructed a rule-tree representation of the target proof which, upon application, will produce a complete target proof that satisfies the target specification, i.e., the transformation is correct (*cf.* **fig.** *4–1*).

We can summarize the passage from source to target by the following steps:[9]

1. The source proof specification is transformed (by partial evaluation) to the target specification.

2. The source rule-tree is abstracted from the source proof.

3. The target rule-tree is constructed through the mapping and transformation of the source rule-tree.

4. The target rule-tree is applied to the target specification thus producing the target proof.

5. The target (specialized) algorithm is extracted from the proof.

The whole process, from 1 to 5, is completely automatic.[10]

Explanation of the Source Rule-Tree for Sumlist

The main goal to be proved is the following statement of the sumlist theorem (where we use *list* as shorthand for the type *nat list*):

$$\forall l\!:\!list\ \exists z\!:\!nat\ sumlist(l) = z,$$

which, together with the relevant lemmas,

$lemma\ 1:\ sum([]) = 0,$ and

$lemma\ 2:\ \forall l : list.\ \exists k : nat.\ \exists hd : nat.\ \exists tl : list.\ l = hd :: tl \wedge sum(tl) = k \rightarrow\ sum(l) = hd + k,$

forms a complete specification for the (source) sumlist program.

The (source) proof of this specification is displayed in the next section. The source rule-tree abstracted, by the PSS, is shown below in **fig.** 4–2, where the

---

[9]Following the *sumlist* example, we flesh this process out into 13 steps and provide more detail concerning the rule-tree manipulations.

[10]The PSS is, however, provided with a "chatty mode" such that the user can, if she or he desires, observe the results of each of 1 to 5.

```
[intro then
        [elim(l) then
                [intro(0) then lemma 1 then wfftacs,
                 intro(explicit(v2 + v0)) then lemma 2 then wfftacs
                ],
        ],
]
```

Figure 4–2: The rule-tree for the source *sumlist* proof

recorded rules (refinements and tactics) have the following purpose:

- The application of the *intro* tactic strips off the universal quantifier from the top level goal, which has the effect of splitting the conditional into assumption and (sub)goal.

- The application of the *elim* tactic on l performs *list* induction on the list l.

- The application of $intro(0)$ provides the base case value – the sum of the integers in the empty list.

- For the step case we introduce, by the $intro(explicit(...))$ tactic, the recursive value as the sum, $v2$, of the integers in the tail of the input list plus the head of the input list $v0$. This corresponds to adding the induction hypothesis to the induction variable thus $v2 + v0$.

- The first application of the $wfftacs$[11] tactic is used to check that $v2 + v0$ inhabits the basic *integer* type (*int*). That is, that the sum of two integers is itself an integer.[12]

---

[11]The $wfftac$ tactic is composed of numerous smaller tactics, and refinements, which, depending on the context, satisfy various type checking and well-formedness goals.

[12]There would be no problem with specifying the input and output of our source specification as inhabiting the basic type *nat* (*cf. Chapter 2*).

- The second application of the $wfftac$ tactic is used to state a type for the input list l. In this case that it is in the (lowest level) universe $u1$.

The Source Proof

The source proof for the *sumlist* function, from which the source rule-tree is abstracted, is schematically displayed in **fig.** 4–3. Refinements that witness a value, v, for an existentially quantified variable are depicted by $\exists - intro(n)$. This is the equivalent of $intro(explicit(n))$ in the rule-trees (*cf.* **fig.** 4–2).

The (Automatic) Construction of the Target Rule-Tree

The construction of the *sumlist* target specialized proof and of the automatic specialization mechanism is best described if we do a step by step explanation of what has to be done to the source proof in order to map it to a target which takes a list of specific length and performs *sumlist* on that.

We shall break up the explanation into stages. The first stage consists of obtaining a (specialized) target specification from the (non-specialized) source specification. The target specification is not entered into the rule tree, but recorded so that, upon termination, the completed target rule-tree can be applied to it thus producing a completed target proof.[13]

The remaining stages consist of describing how the target rule-tree is constructed from the source rule-tree. At each stage we explain what rules are entered into the developing target rule-tree, and the effects that result from the application of those rules when the completed target rule-tree is applied to the target specification.

Throughout the explanation n will refer to the length, number of elements, of the input list. For the specialization n is set to 3. That is, we wish to transform

---

[13]Recall that the rule-tree abstractions are akin both to a large OYSTER tactic, which combines a number of proof rules, and to a skeleton of a proof in which the inference rules of the proof, but not the formulas to which they are applied, are recorded.

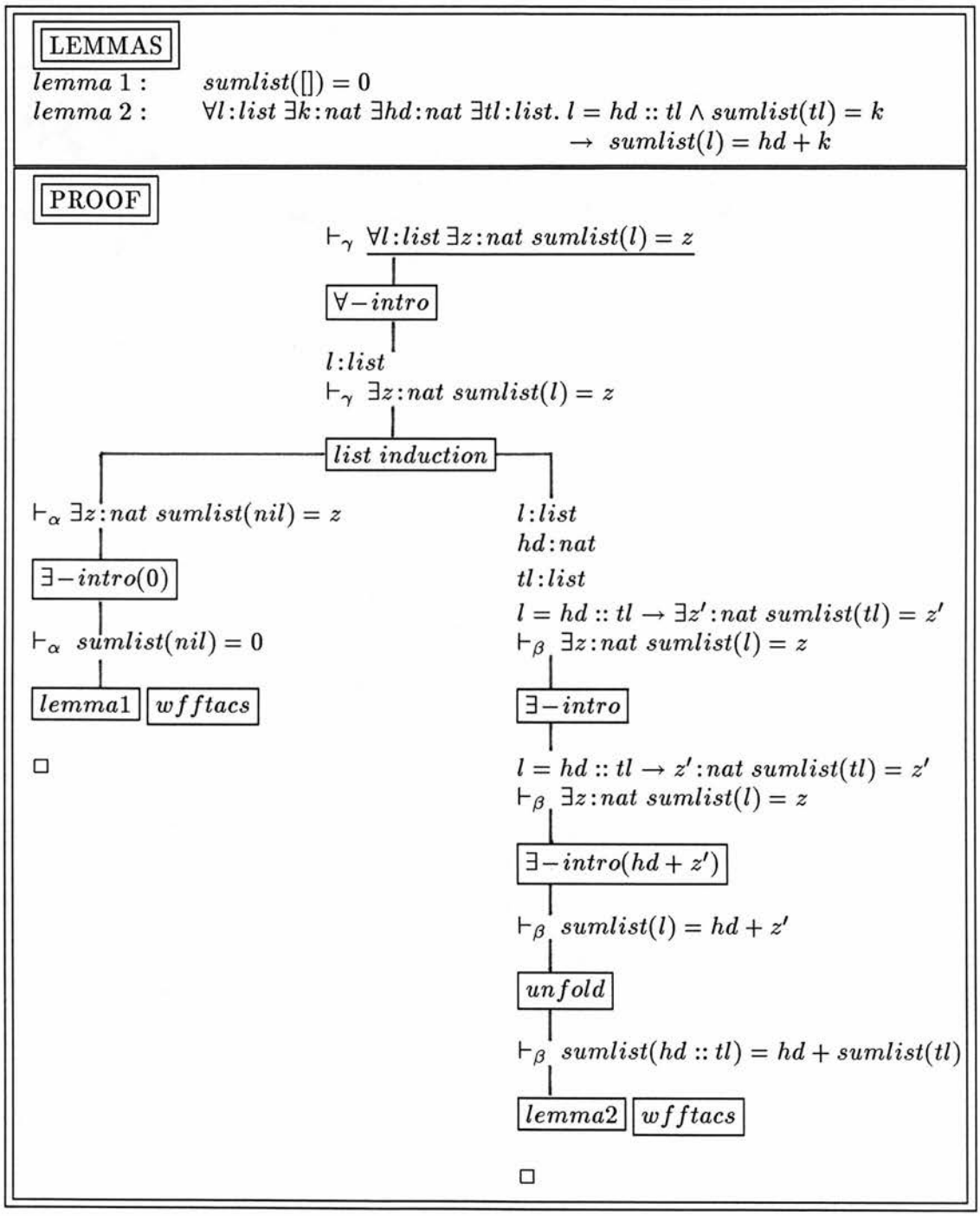


Figure 4–3: Source synthesis proof for the *sumlist* program

the source proof, satisfying a specification which takes an input of any length n, to a target proof satisfying a specification which takes as input the specific input list length of 3 and outputs the sum of those 3 integers. Of course, n is not limited to 3 and can be set to any desired specialization.

STAGE 1: *Transforming the Specification*

For our example, where the specialized length is 3, we require mapping procedures which transform the complete source specification:

sumlist source spec: $\forall l\!:\!list\,\exists z\!:\!nat.\ sumlist(l) = z$,

into a (specialized) complete target specification:

sumlist target spec: $\forall a\!:\!nat\,\forall b\!:\!nat\,\forall c\!:\!nat\,\exists z\!:\!nat.\ sumlist([a,b,c]) = z$,

where *sumlist* is defined through the lemmas in **fig.** 4–3.

Clearly, the transformation on the source *sumlist* specification consists in retaining the specified source output, by a one on one mapping, and replacing the source input, $l\!:\!list$, by a list containing the specialized number of elements, $[a,b,c]$, where each of a,b, and c are specified, in the target specification hypothesis list, as natural numbers. So, restricting ourselves for the present to programs that take some list, l, as input, and output an object of some type **Type** – where **Type** can be considered as a meta-variable ranging over types – then given any value of n as our specialization on the length of l we can schematically represent the transformation of any such source specification to the specialized target specification as follows:

SOURCE: $\forall l\!:\!list\,\exists z\!:\!\mathbf{Type}.\ P \rightarrow f(l) = z$,

TARGET: $\forall a_1\!:\!\mathbf{Type}\,\forall a_2\!:\!\mathbf{Type}\,\forall a_3\!:\!\mathbf{Type}\ldots\forall a_n\!:\!\mathbf{Type}.$
$$P' \rightarrow f([a_1,a_2,a_3,\ldots,a_n]) = z,$$

where n is a number designated by the user when he applies the specialization, and P represents other additional preconditions, if any, of the main goal (these may contain further quantified terms which may also be subject to specialization

if they contain any reference to the input list, l, being specialized, hence the P' in the target specification).

Note that a property of specialization is an *increase* in the specification content corresponding, in the case of programs that take lists for inputs, to specifying the length of the input.

STAGE 2:*Mapping and Transforming* $\forall -$*Intro and* Type Checking *Tactics*

The next stage consists of the following mappings on the source proof:

- Mapping across the *intro* rules. These split up the main goal into the separate preconditions and then adds them to the hypothesis list.

- Mapping across the corresponding type checking rules. That is, ensure that the target proof, albeit partial, is well-formed.

In the source proof, **fig.** 4–3, which is specified for a *single* input *list*, only one $\forall -$*intro* rule application is required to split the sequent into assumption and (sub)goal.

In the target we require n applications of the $\forall -$*intro* rule in order to separate

$$\forall a_1 : \mathbf{Type}\ \forall a_2 : \mathbf{Type}\ \forall a_3 : \mathbf{Type} \ldots \forall a_n : \mathbf{Type}$$

into the separate constituents.

Each application of an *intro* rule is accompanied by the corresponding type checking rule applications. So transforming that portion of the source proof consisting of the application of $\forall -$*intro* corresponds to the following:

- If the *first* rule application of the source proof is $\forall -$*intro* then iteratively apply $\forall$–*intro* n times to the target specification. For each sub-goal produced of the form X *in* $u(Y)$ do a well-formedness check (i.e., provide the type which matches X with a universe).

After the $\forall - intro$ applications are applied to our example target specification, with $n = 3$, we will arrive at the following target proof node (exactly 3 refinement levels deep):

refinement :	$\boxed{3 \times \forall - intro}$
hypotheses :	$a{:}nat\ b{:}nat\ c{:}nat$
conclusion :	$\vdash_{\gamma} \exists z{:}nat\ sumlist([a,b,c]) = z$

where a, b and c are the separate conjuncts of the target specification antecedent (preconditions) and, the satisfaction of the sub-goal requires the introduction of an object of type nat.[14]

It should be noted that *type-checking* rules may differ considerably between source and target. This is to be expected since in the source we are dealing with a single object l of type $nat\ \ list$, whereas in the target we have n objects each of type nat. This does not present a problem for the specialization system since *all* type-checking, and most well-formedness goals, can be satisfied by applying the proof tactic $wfftacs$.

STAGE 3: *Grounding Induction Schemata*

The whole rationale of specializing induction schemata means that we do not require a general list induction, *elim(l)*, as we did in the source because our target input and output lists are not general, they are of fixed length n.

In effect, we require a target proof construct that mirrors the application of *list induction*, on the list l in our source proof when the input is fixed at n elements. The $elim(l)$ rule application in the source proof results in two sub-goals, one

[14]Since we are *synthesizing* a program from a main goal which is limited to simple *typing* information on the input and output, then sub-goals, such as the *nat* above, generally require *any* new program construct, as long as it is of type *nat*, to be entered into the proof, rather than proving that some computational property holds.

corresponding to the base case and the other to the step case. We are also provided with three new assumptions: the head of l in *nat*; the tail of l in *natlist*; and the answer for *sumlist* tail of l (i.e., the induction hypothesis).

The Rôle of the Rule-tree

A transformation tactic, *ind_trans*, accesses that portion of the source rule-tree corresponding to the application of list induction. This will contain the refinements responsible for witnessing the induction base and step cases, the *witnessing refinements*, and the subsequent refinements responsible for verification (in our example these will be lemma applications and well-formedness checks). This is done by searching for the sub-list structure within the rule-tree headed by an application of the *elim* rule on the input list l. We display this portion of the source rule-tree in **fig.** 4–4 below,[15] where $v2$ is the induction hypothesis, corresponding to an output for the tail, tl, of the input list, i.e., $sumlist(tl(l))$, and where $v0$ is the head of the input list, i.e., $hd(l)$. From this construct, *ind_trans* isolates the

$$
\begin{array}{l}
[elim(l)\ then \\
\quad [intro(0)\ then\ lemma\ 1\ then\ wfftacs \\
\quad intro(explicit(v2 + v0))\ then\ lemma\ 2\ then\ wfftacs \\
\quad] \\
]
\end{array}
$$

Figure 4–4: Rule tree construct corresponding to application of source induction

rules that refine the induction base and step cases, and then forms a record of the base and step witnesses, and the respective lemmas used to verify the witnessed cases as follows:

base witness: $intro(0)$;

step witness: $intro(explicit(v2 + v0))$;

---

[15]Note that **fig.** 4–4 corresponds to the source rule-tree, depicted in **fig.** 4–2, without the initial *intro* rule.

base verification: *lemma* 1 *then* $wfftacs$; and

step verification: *lemma* 2 *then* $wfftacs$.

From this information the transformations produce a target (rule-tree) nested list structure $n+1$ levels deep, where n is the desired specialization on the length of l. This nested list structure is entered into the target rule-tree (after the initial *intro* rules of STAGE 2).

Upon application, this target (rule-tree) nested list structure will produce a target (proof) tree structure consisting of n nested cuts, where a cut is introduced into the proof by the *sequence* rule (appearing as *seq* in the OYSTER proofs and OMTS rule-trees). We shall refer to such a rule-tree structure as the *grounding rule-tree sub-list*. The resulting proof structure, corresponding to an instantiated unfolding of the source induction schema when the input length is n, we refer to as a (source) *grounded* induction schema.

Recall from *Chapter 2*, that applying the *seq* rule at a proof node N, with a corresponding goal G, has the effect of introducing a new fact, H, into the proof by the introduction of a two new nodes, $N1$ and $N2$, with two corresponding new subgoals $G1$ and $G2$, where $G1$ is responsible for proving H, and where $G2$ represents the original subtree with H as an additional hypothesis (thus $G = G2$ and $G1 = \vdash H$).

For our example, n was set to 3, and the *grounded* induction schema is (automatically) produced by applying the below target rule-tree, **fig.** 4–5, to the target specification (where an explanation of the recorded rules, and of the automatic construction of the target rule-tree, follows). The boxed sub-list corresponds to the grounding rule-tree sub-list. Regarding the grounding rule-tree sub-list, each *seq* application produces the two further sub-goals where the first, $G1$, requires that we specify what our newly sequenced in object, $new[z_2]$, is, and the second, $G2$, requires that we provide a witness for z_1 in terms of z_2.

Letting uppercase represent meta-variables, then we can schematically represent the relation between N, $N1$, and $N2$ in the target proof by **fig.** 4–6 below,

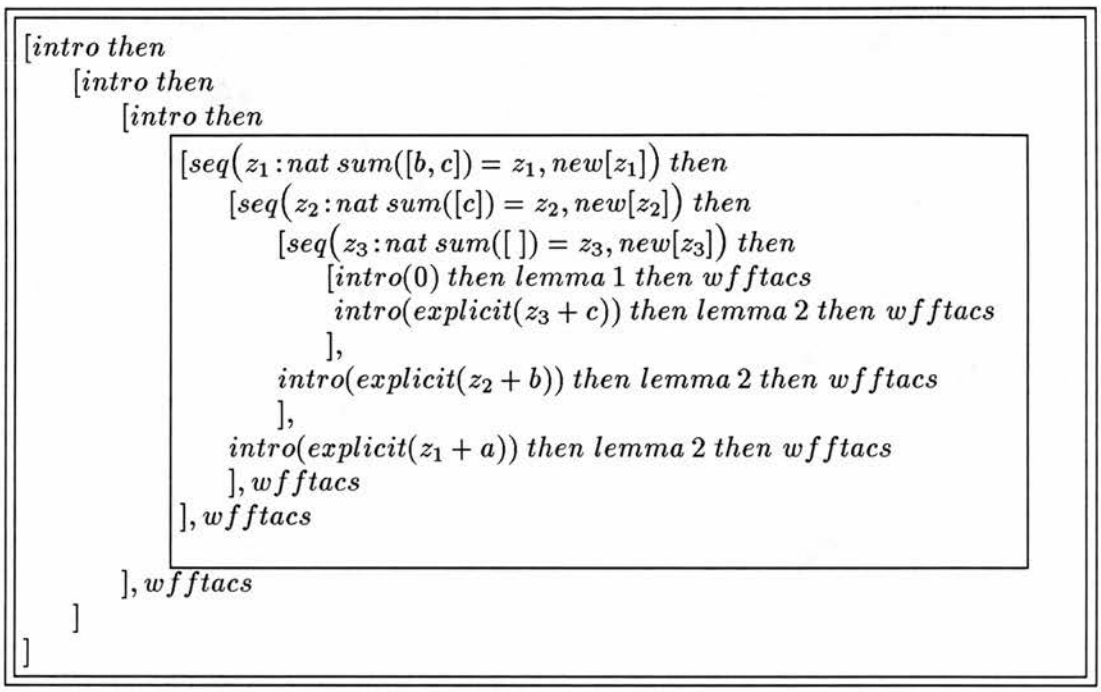


Figure 4–5: The target rule-tree for $sumlist([a, b, c])$

where L_2 is the tail, tl, of L_1, i.e., $L_2 = tl(L_1)$, such that, if $hd(L_1)$ is the head, hd, of L_1, then $sumlist(L_1) = (sumlist(L_2) + hd(L_1))$).

The Refinements and Transformations Required to Satisfy the Cut (Sub)Goals *G1* and *G2*

We now consider, taking each of $G1$ and $G2$ in turn, what is required to satisfy each of the cut sub-goals *and* the transformations performed by the PSS, on source and target proof constructs, in order to achieve the satisfactions.

So, letting the progression $1, ..., i, ...n$ represent the successive levels of the grounded induction, then, regarding the sequencing at node N_i, the first sub-goal is satisfied, in each case, by sequencing in a further $Z_{i+1} : nat\, sumlist(L_{i+1}) = Z_{i+1}$, where $L_{i+1} = tl(L_i)$, until eventually, depending on n, we reach the base case counterpart where our input will be [ ] (i.e., $L_n = [\,]$). The base case output (witness), 0, is simply mapped over from the source (i.e., $Z_n = 0$), and then verified by appealing to the source base lemma, *lemma* 1.

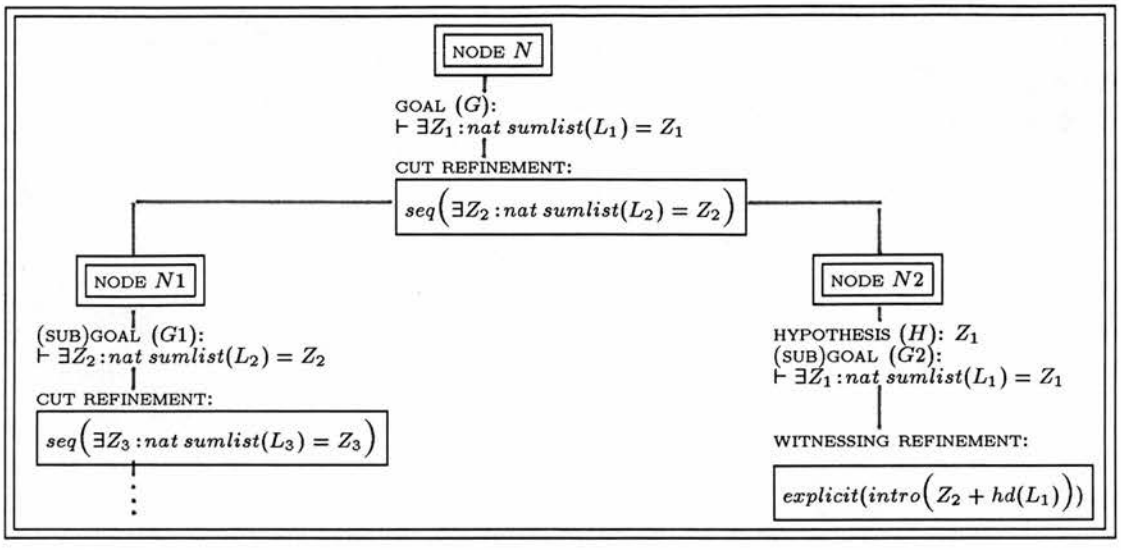


Figure 4–6: A schematic representation of the induction grounding cut refinements

The PSS obtains the correct sequencing steps at each level by performing transformations on the target main goal (specification) $\vdash\ z : nat\ sumlist(l) = z$. This simply involves substituting the input l for $tl(l)$, and substituting the output z for z_1, to obtain the first sequencing: $seq(z_1 : nat\ sumlist(tl(l)) = z_1)$. The system then substitutes the input l for $tl(tl(l))$, and substitutes the output z for z_2 to obtain the next sequencing: $seq(z_2 : nat\ sumlist(tl(tl(l))) = z_2)$. The process continues until the substituted value for l is [] and that for z is z_n, whereupon the source base case witness is mapped across one on one.

The second subgoal is satisfied in each case by actually providing a witness in terms of the outputs $z, z_1, ..., z_n$ of the sequenced in formulae at the first sub-goal. The PSS obtains the correct witness, at each level of the grounded induction, by performing transformations on the witnessing refinement, $intro(explicit(v2+v0))$, at the source step case: each witness is as a result of substituting, in turn, each of the n input list elements for $v0$, and each of $z, z_1, ..., z_n$ for $v2$ where, as (the source) list induction dictates, $v0$ is the head of the input list at each unravelling, or unfolding, of the induction step case, and $v2$ is the output for *sumlist* when the input is the tail of the list at each unfolding.

The Target Proof for the *sumlist* Induction Grounding

Each time a portion of the source rule tree is transformed and mapped into the developing target rule tree the system ensures that, upon application, it will result in a viable partial proof (i.e., a proof of status *partial* as opposed to *bad* thus signifying that all rule applications are legitimate). This is done simply by having the system *mark* and then *copy* each new development of the target rule-tree: OYSTER accesses the (partial) rule-tree and then attempts to apply it. If the application is successful the system will proceed with the next mapping. However, for efficiencies sake, this approach is optional: if one is particularly confident that the transformations will not result in an invalid proof and therefore do not wish to waste time continually *marking* and *copying* proof branches then there is a mechanism setting which only *marks* and *copies* the target proof once it is complete.

Once the source rule-tree transformation has terminated, the completed target rule-tree will produce, upon application to the *sumlist* target specification, the target proof shown in **fig.** 4–7 (where *sumlist* is abbreviated to *sum*, and the lemmas are the same as for the source proof). The proofs produced for specializations greater or less than $n = 3$ will be essentially the same, except that the nested sequence of cut applications will be correspondingly deeper or shallower.

Summerizing Chart for Induction Grounding Specializations

We can represent the control flow for the specialization on the *length* of an input list by the sequence of 13 operations below. Note that the source specification need only have the minimal condition, as does the sumlist proof, that the input is of type *list*. Subsequently, we provide the necessary modifications to steps 2, 3, and 8 required such that this minimal condition need not apply. We omit the frequent application of type checking rules which chiefly consist of mapping across the source type-checking rules, and adapting them accordingly, to apply to *each* of the newly introduced objects in the target specification.

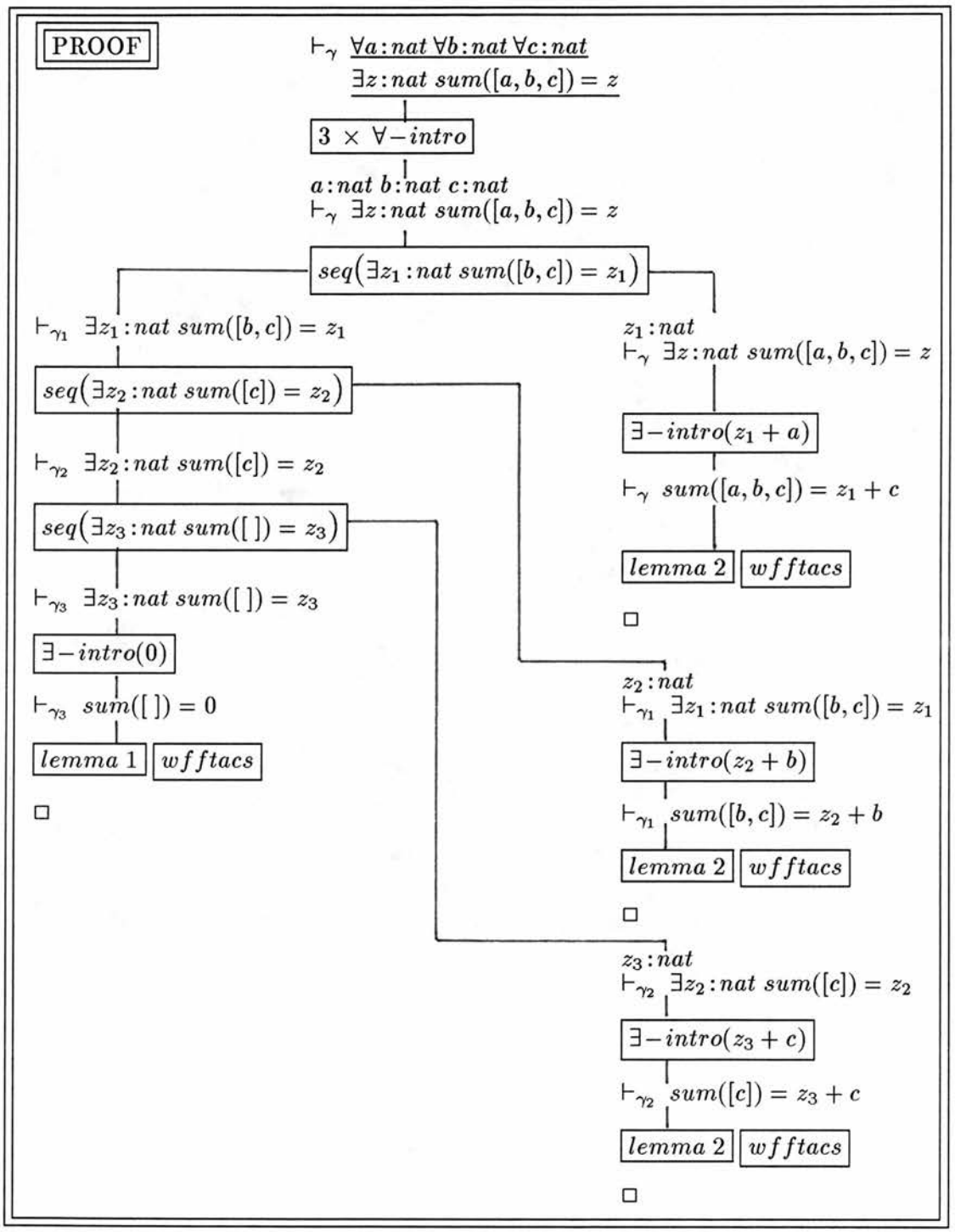


Figure 4–7: Target synthesis proof for the *sumlist* (abbr. *sum*) program

1. Access source proof and convert into *rule tree*, R.

2. Access, within R, source specification S_{spec},

$$\forall l{:}list\, \exists z{:}\mathbf{Type}.\ P \rightarrow f(l) = z,$$

and transform into target specification, T_{spec}, according to value, n, of specialized parameter:

$$\forall a_1{:}\mathbf{Type} \forall a_2{:}\mathbf{Type} \forall a_3{:}\mathbf{Type} ... \forall a_n{:}\mathbf{Type}.\ P' \rightarrow f([a_1, a_2, a_3, ..., a_n]) = z.$$

3. Map across, one on one, the source application of the $\forall - intro$ rule, and apply n times application of $\forall - intro$ to split T_{spec} (if the source contains m $\forall - intro$ prior to that on the induction candidate – the list to be specialized – then apply $m + n$ applications of $\forall - intro$).

4. Sequence in n new facts, $v_1, v_2, v_3 ... v_n$ into proof.

5. Access source Induction Schema, I (search for sub-tree whose root node is an application of the induction rule, $elim(X)$ where X is the induction variable).

6. Access, within I, the base case, I_b, and step case, I_s.

7. Prune induction case split. Replace by:

8. Nested structure, $n + 1$ levels deep, where each of the n levels contains the subgoal achieved by *substituting* each of $a_1, a_2, a_3, ..., a_n$, and $v_1, v_2, v_3, ..., v_n$ into the source proof step case I_s.

9. The $n + 1$ level corresponds to the base case value of the source function. Map identically from source.

10. Check for further *non-nested* induction schema (as in stage 5). If present then check if induction is on the list l. If so, then specialize by stages 5-9.

11. Check for *nested* induction schema, search for application of induction, $elim(l)$, *within* sub-proof-tree headed by the node accessed at stage 5. If nested schema present then specialize by stages 5-9.

12. Apply new target rule-tree to T_{spec} to produce target proof T_{proof}.

13. Extract, by the OYSTER extraction process, the target program.

Induction Grounding on Schemas Other Than List Induction

We have dealt mainly with induction grounding on proofs that employ *list* induction since this incorporates all the key steps of induction grounding on, say, *stepwise* inductive proofs (over natural numbers or integers). Induction grounding on inductions which are not performed on lists follows the same general strategy, although there are a few differences.

The modifications we need to make to the above 13 steps if induction grounding is performed on a source proof of the following (schematic) specification:

$$\forall y{:}\underline{nat}\,\exists z{:}\mathbf{Type}.\ P \rightarrow f(y) = z,$$

are to steps 2, 3, and 8:

2'. Access, within R, source specification S_{spec},

$$\forall y{:}nat\,\exists z{:}\mathbf{Type}.\ P \rightarrow f(y) = z,$$

and transform into target specification, T_{spec}, according to value, n, of specialized parameter:

$$\exists z{:}\mathbf{Type}.\ P \rightarrow f(n) = z.$$

3'. There is no real counterpart to 3, since by specializing the source specification we remove the universal quantifier binding the parameter which is instantiated in the target (as n).

7'. Prune induction case split. Replace by:

8'. Nested structure, $n+1$ levels deep, where each of the n levels contains the subgoal achieved by *substituting* each of $n, n-1, n-2, n-3, \ldots$ into the source proof step case I_s.

Recalling *Chapter 2*, §2.3.3, step 8' corresponds to the replacing of an infinite sequence of "sub-proofs", implicit within stepwise induction, by a nested application of cuts which sequence into the proof the first n "sub-proofs" of the infinite sequence (we shall elaborate on this shortly).

The Target (Extract) Algorithm

For our example target, where $n = 3$, the extract term for the target will be as follows:

$$\boxed{\lambda a,\ \lambda b,\ \lambda c.\,(\lambda v2,(v2+a)(\lambda v2,(v2+b)(\lambda\ v2,(v2+c)(0))))}$$

Note that there is no recursion schema in the target extract. It has been transformed, via the proof pruning, from a recursive procedure that operates upon lists of *any* length (i.e., all objects of type *list*) into a non-recursive procedure procedure that operates on a specific range of lists (i.e., that range determined by n). Unlike the source extract, the target extract is not required to recurse down an input list of arbitrary length, adding the first element to the remainder at each pass, but is rather tuned to make only $n + 1$ calls to the addition function (i.e., $n + 1$ calls to $X + E$, where X and E are meta-variables in the *lambda* expression $lambda(X, X + E)$).

Efficiency Comparisons of Target and Source Proof Extracts

The efficiency of each proof extract is measured by a procedure which takes the average run time over some large, user-specified, number of applications of the extracts (this is to account for fluctuations in CPU time). With a setting that averages over, say, 100 runs, the results are as expected. The extract of the specialized target is faster to a degree of 1.4 than the source proof. This is because in the source the step case value for the induction has to be calculated at each unfolding whereas with the target these values are provided hence cutting down the amount of computation:[16]

source sumlist extract: Average CPU time for 100 runs = 326.7 ms.

target sumlist extract: Average CPU time for 100 runs = 242.6 ms.

---

[16]Later we provide an analysis of the complexity associated with the respective extracts.

The target extract can be collapsed further by partially evaluating the extract. This is done, somewhat trivially, by the OYSTER $eval(X, Y)$ predicate (§2.2.1). If we partially evaluate our target *sumlist* extract in this way, we obtain the following partially evaluated extract:

$$\boxed{\lambda a,\ \lambda b,\ \lambda c.\ (a + b + c)}$$

then we remove all computational effort associated with evaluating the base case value. Consequently, the resulting run time, an average CPU time of 198.4 ms., is comparatively faster than the non-partially evaluated extract.

Observations

The following points are worth noting:

- We noted at the outset, §4.1.3, that induction grounding is a form of pruning, although different to both normalization and dependency pruning. It is also not an example of the type of pruning discussed in, for example, (Bruynooghe *et al.*, 1989; De Schreye & Bruynooghe, 1989) where partial evaluation of a *program* guides the pruning of *identical* sub-computations. Rather, the pruning is governed by knowledge concerning induction schema, and its duality with the extracted recursion schema, such that recursive programs are transformed to non-recursive programs that operate more efficiently on the specified specialized input(s).

- The transformation is correctness guaranteed in the sense that the target proof specification completely specifies an input list containing the desired, specialized, number of elements, and the output is specified as the sum of those elements.

- By specializing induction schemata we remove recursion from the corresponding program constructs. In effect by *pruning* the induction case split from the source proof, we remove the recursion schema from the extract. The *list_ind* schema in the source extract has been replaced by the specific

unfolding of that schema when, for example, the input is of length 3. This is done by pruning the induction case split from the source proof and then sequencing in what amounts to an instantiated schema with the length of l, being set to 3 (or any desired length, n).

- There is an inverse relationship between the complexity of proof and program: the specialized proof will be syntactically more complex than the source proof.

  However, the extract program will be computationally less complex since much evaluation will have been taken care of.

- Regarding induction grounding, it is impossible for the target program to satisfy the same specification as the source since the induction grounding transformations remove the recursion schema, which operates over all objects of a specific type, and replace it by a non-recursive structure that is tuned to operate on a specialized sub-set of those objects (for example, lists of a specific length). There is no way this can be done without placing extra conditions on the input, i.e., without specializing the original source specification such that, for example, the input does not take *any* input list, but rather any list containing the specialized number of elements.

  However, the target program of an induction grounding transformation *is* correct with respect to it's own *complete* target specification ((i) above): the specification totally and unambiguously captures the desired adaptation – specialization – of the source input-output relation.

  So in this sense, the presence of a target proof, from which the target program is extracted, ensures that induction grounding satisfies the usual correctness criteria for program transformation, i.e., the target is correct with respect to the desired (specialized) input-output relation.

Induction Grounding $\simeq$ Application of Cut Elimination Theorem

The *induction grounding* is an example of *recursive to non-recursive* program transformation through the partial evaluation of proofs. It is, in effect, a *reverse* application of the *cut elimination theorem*, (Szabo, 1969), in that every proof in a Σ_1 induction system can be reduced to one with only Σ_1-cuts.[17]

Induction grounding consists in eliminating the implicit infinite unraveling of the induction schema in favour of a (partially) evaluated finite structure consisting of $\Sigma_1 - Ind$ quantified formulae — specifically, n nested induction cuts where n is bound to the specialization, i.e., $\Sigma_1 - Ind$ induction schemata are replaced, through partial evaluation, with Σ_1-cuts.

Consider the following simple induction schema over the naturals:

$$\mathbf{R}: \ (P(0) \wedge \forall n'(P(n') \rightarrow P(s(n')))) \rightarrow \forall n P(n).$$

Clearly, it is a defining property of such a schema that the variable n ranges over *all* the naturals. Depending on the particular input value for n, the recursive program construct extracted from **R** would have to be evaluated (unpacked or unraveled) appropriately.

Dual considerations apply to partially evaluating induction schemata, if we have a proof of $P(n)$ by induction then we can unravel it into an infinite sequence of "sub-proofs":

$$P(0), P(1), P(2), ..., P(n), P(n+1), ...,$$

where *each* $P(n)$ requires n nested *induction cuts*:

$$P(0), P(0) \longrightarrow P(1), P(1) \longrightarrow P(2), ..., P(n-1) \longrightarrow P(n), P(n) \longrightarrow P(n+1), ...$$

---

[17]The Σ_1 induction theory is a subset of Martin-Loff Intuitionistic type theory, the important property in common being *quantification*. We distinguish induction on an existential formula:

$$\Sigma_1^0 - \textit{induction rule}: \ \frac{\vdash P(x)}{\vdash P(0) \qquad P(v) \vdash P(s(v))} \quad \textit{where } P = \exists y R(x,y),$$

from induction on a universally quantified formula, $\Pi_2^0 - \textit{induction}$, where $P = \forall x \exists y R(x,y)$.

So, granted the provision for *infinite* proof-rules, we could, in theory, eliminate the induction completely and replace it by:

$$(P(0), P(1), P(2), ..., P(n), P(n+1), ...) \longrightarrow \forall x P(x).$$

In effect, the specialization of such a schema removes, through partial evaluation, all the evaluation associated with such infinite unraveling *before* the (specialized) program is run. It does so simply by substituting the induction schema for a finite tree structure consisting of successive unravelings of **R** with the desired value for n. So, for example, with n set to 4 we obtain the following target *grounded* induction, **G**:

$$\mathbf{G}: \quad P(0), P(0) \longrightarrow P(1), P(1) \longrightarrow P(2), ..., P(3) \longrightarrow P(4).$$

So, by grounding the induction we also remove the dual recursion.

The Inverse Complexity Relation

Induction grounding illustrates the inverse relationship between the complexity of proof and program, the specialized proof will be syntactically more complex than the source proof. This is because the infinite sequence of "sub-proofs" associated with a proof of $P(n)$ by induction are, as far as the synthesis is concerned, implicit, and an induction proof tree will have a branching structure determined by the number of induction cases.[18] The grounded inductive proof tree will, on the other hand, have a branching structure n levels deep, where n is the desired specialization. So although, in CPU terms, the specialized algorithm is more efficient than the source algorithm, when running on the input n, the specialized proof has a more complex branching pattern than the source proof.

Or more intuitively speaking, since the partially evaluated source specification has a greater content than the non-partially evaluated source specification then

---

[18]In our example, stepwise induction was employed which has two induction cases. For a clear and informative investigation into the complexity of inductive proofs, and the recursive programs that they synthesize, the reader should consult (Wainer, 1990) .

the former will require more theorem proving than the latter. This extra theorem proving, albeit automatically performed by the PSS, pays off in that the algorithm extracted from the specialized proof is tuned to operate more efficiently on that particular input which accounted for the additional specification content in the first place.

4.2.3 Example 2: The Specialization of Nested Induction Structures

The *sumlist* example was good for explanation's sake but not particularly practical. A more practical case for specialization would be the specialization of a sorting algorithm. This would consist in transforming a general sorting algorithm into a specialized analogue which sorts lists of a specified length i.e., sorting a specified number, n, of elements.

The mechanisms described above for specializing relatively simple proofs such as the *sumlist* function will also specialize the sorting algorithms. The purpose of briefly describing a further example of induction grounding is twofold, firstly, to illustrate that the PSS is capable of specializing nested recursion schemata through grounding nested induction schemata, and secondly, to illustrate, and discuss the ramifications of, the PSS transformations on proofs from weak specifications.[19]

Transforming Programs Synthesized from Weak Specifications (Heuristic Specialization)

By employing weakly specified source proofs we do, of course, lose the correctness criteria and so such transformations should be regarded as a heuristic process. The specialization of the insertion sort synthesis proof qualifies as such a *heuristic specialization.* The target algorithm will, nevertheless, satisfy the partially

[19]Recalling § 2.2.9, a weak specification is an incomplete specification that only captures the typing properties of a programs input/output.

evaluated source specification and so the user can, in such cases, have the same expectation that the target computes the desired input/output relation as he or she has concerning the source algorithm.

In all essentials, the methodology for specializations on weakly specified proofs is the same as for specializations on completely specified proofs. However, an interesting property of specializing programs extracted from proofs with incomplete specifications is that by only specifying the *type* of the input and output – e.g., that the input is a list and the output an integer – the methodology employed for the such examples is general regarding:

- *any* other program synthesized from the same weak specification (and there are infinitly many programs which are satisfied by such weak specifications); and

- *any* other program for which the complete specification falls within the scope of the weak specification. By this we mean that the content of the complete specification includes that of the weak specification. So for example if we can specialize on the length of an input list, l, where the source proof has the following weak specification:

  $$\forall input : nat\ list \rightarrow \exists output : nat\ list,$$

  then we can also specialize on the length of an input list, l, where the source proof has the following complete specification for a naive (inefficient) sorting algorithm:

  $$\forall l_1 : nat\ list\ \exists l_2 : nat\ list\ ordered(l_2) \wedge permutation(l_2, l_1) \rightarrow sorted(l_2),$$

  where *permutation* takes a list l_1 as input and produces a list, l_3, of lists consisting of all the permutations of l_1, and where *ordered* selects the ordered permutation, l_2 amongst l_3.

This is a nice feature of transforming OYSTER proofs since it allows us to adapt, or place conditions upon, the *typing* properties of a specification. This is not possible with Goad's system since there is no target proof, and consequently no target specification which would specify the type of the specialized output.

Specializing *Insertion Sort* (*insort*)

As a space saving device we shall, for the purposes of this example, only reproduce the rule-trees and extracts.

For the *insertion sort* proof, the source extract defines an algorithm which sorts *any* number of elements by recursively inserting the head of the list into the tail at the correct *ordered* position (the sorted list being constructed travelling up the recursion). The weak specification for the *insertion sort* proof, *insort*, merely specifies the typing restrictions on the (source) programs input and output thus:[20]

SOURCE SPEC FOR *insort*: $\forall l{:}nat\,list \rightarrow \exists z{:}nat\,list.$

That is, for all inputs of type *list* there exists an output also of type *list*.

Note that the weak specification contains no reference to sorting, or indeed to any function, the onus remaining very much with the human synthesizer to construct an algorithm with the desired functionality. Unlike the previous *sumlist* example, the source proof for *insort* will consist primarily of the synthesis component, with no reference to lemmas that verify the witnessed induction steps.

The lack of verification content is reflected in the source rule-tree abstraction, which also contains no record of any verification lemmas. This does not mean that the proof is ill-formed, but that there is no logical guarantee that the extract will compute the *desired* function (hence the heuristic nature of specializations on such source proofs).

The extract for the source proof of insertion sort is as follows:

$$\boxed{\lambda l.\ list\_ind(l, nil, [v0, v2, term\_of(insert)(v0)(v2)])}$$

where $term\_of(insert)$ is an auxilliary call to the *insert* function which inserts the head, $v0$, of the (recursive) tail $v2$, at the correct sorted position amongst the elements sorted so far.

---

[20]The PSS will also specialize a completely specified proof for *insort*.

```
[intro then
        [elim(l) then
                [intro(nil) then
                        [wfftacs],
                 intro(explicit(term_of(insert) of v0 of v2)) then
                        [wfftacs]
                ],
         wfftacs
        ]
]
```

Figure 4–8: The rule-tree for the source *insort* proof.

The rule-tree abstracted from the source proof is depicted in **fig.** 4–8, where the recorded induction case witnessing rules are as follows:

base witness: $intro(nil)$; and

step witness: $intro(explicit(term\_of(insert)\ of\ v0\ of\ v2))$.

The target proof will be a specialization of the source with n set this time to, say, 2. So the target extract will be specialized to sort pairs. Hence, by the same methodology as for the *sumlist* example, the mapping procedures will transform the source specification to the following target specification:

TARGET SPEC FOR *insort*: $\forall a{:}nat\,\forall b{:}nat \rightarrow \exists y{:}nat\ list.$

which, albeit still weak, contains more content than the source specification (corresponding to the desired specialization on the input).

Through the induction grounding transformations performed on the source rule-tree, the induction schema is pruned away and replaced by $n = 2$ nested sub-goals. This is done, in a similar fashion as for the *sumlist* example, by sequencing into the proof $n = 2$ new sub-goals, via applying the corresponding *seq* refinements in the target rule-tree. The sequenced sub-goals are again attained by performing substitutions on the target main goal, to state the existence of the newly entered fact, and by substituting in turn one of the $n = 2$ new objects, a and b into the source step case, to provide a witness for each new fact.

The grounded induction schema appears as the boxed (sub)list of the representation of the target rule-tree abstraction depicted in **fig.** 4–9. Note that, unlike the *sumlist* example, each sequenced new fact does not contain any reference to the actual function being computed. This property is inherited from the weak specification of the target, which is in turn inherited from the (transformations on the) weak source specification.

```
[intro then
   [intro then
      [seq(z1 : nat list, new[z1]) then
         [seq(z2 : nat list, new[z2]) then
            [intro(0) then wfftacs
            intro(explicit(term_of(insert) of b of z2)) then wfftacs
            ],
         intro(explicit(term_of(insert) of a of z1)) then wfftacs
         ], wfftacs
      ], wfftacs

   ],
]
```

Figure 4–9: The target rule-tree for $insort([a, b])$

The program extracted from the proof, that results from applying the target rule-tree to the target specification, is shown in **fig.** 4–10. If we run the (automatic)

$$\lambda a,\ \lambda b.\ (\lambda v2.\ (term\_of(insert)(a)(v2))(\lambda v2.\ (term\_of(insert)(b)(v2))(nil)))$$

Figure 4–10: The target λ-calculus extract for $insort([a, b])$

OYSTER evaluator, *eval*, on this extract, then each of the auxiliary calls to *insert*, within the extract of **fig.** 4–10, unpack to the following, **fig.** 4–11, when the (specialized) input is $[a :: b]$. Decision procedures of the form $less(x, y, v, w)$ are interpreted as $if\ x \leq y\ then\ v\ else\ w$. Hence *less* signifies a case-split in a similar way as does *nat_eq* (§2.2.2). Note that the extract, **fig.** 4–11, although extracted from an induction grounded proof, still contains a recursive structure. This corresponds to the (nested) application of induction used to synthesize the

$$\lambda a,\ \lambda b.\,(less(b, a, b ::$$
$$list\_ind(nil, a :: nil, [v0, v1, v2, less(v0, a, v0 :: v2, a :: v0 :: v1)]), a :: b :: nil))$$

Figure 4–11: The unpacking of the auxiliary *insert* call within **fig.** 4-10

auxilliary *insert* function. Hence there is scope for further induction grounding (see below).

The respective run-times of the source and target are as follows:

source insertion sort extract: Average CPU for 100 runs = 2950.3 ms.

target insertion sort extract: Average CPU time for 100 runs = 2104.0 ms.

Although the specialized proof extract is a significant improvement over the source extract, in terms of run-time, the OYSTER extract evaluation procedure does not provide us with a partially evaluated target extract which marks any significant improvement on the non-partially evaluated extract.[21] This is due to the fact that the partial evaluation of the extract is dependent on the partial evaluation of the sub-theory *insert*. So there is further scope for specialization and this issue is addressed in the following section.

Further Specializing the Sorting Proof: Specializing *Nested* Inductions

The OYSTER proofs corresponding to sorting algorithms are far more complex than the *sumlist* proof, they often contain nested induction schema and/or calls to other proofs. So it would be nice if the specialization could handle the nested inductions and specialize any sub-proofs denoted by the $term\_of(Theorem)$ expression in the

[21] The average CPU time for the partially evaluated target extract, over 100 runs, is 2005 ms.

extract terms above. This would correspond to a *sub-specialization* of the proof for *Theorem*.

This is in fact the case. To specialize sub-proofs and/or nested inductions basically requires a recursive call on the whole specialization mechanism whenever a sub-proof and/or nested induction is encountered during the specialization of the top level proof.

For the *insertion sort* proof this entails specializing the *insert* proof, which is called as a sub proof – $term\_of(insert)$ – in the specialized extract shown above. The full specialization of the *insort* program will, then, involve specializing nested induction schema, albeit indirect, due to the fact that the nested *insert* sub-proof employs list induction in order to insert the head of the list into the tail at the correct *sorted* position.

If the specialization of any nested proof is on the same parameter as with the top level proof then the partial evaluation, n, must be the same in each case. For our example partial evaluation of the top level *insertion sort* proof, the length of input was 2, so this must apply also to the (sub)specialization of the *insert* (sub)proof. Hence as well as having the 2 unfoldings in the displayed extract we also have 2 unfoldings for *each* occurrence of the *insert* sub-proof. Hence the resulting extract from the complete specialized insertion sort proof is pretty large and cumbersome.

The source and target specifications for the *insert* sub-proof are also weakly specified (making no difference to the main *insort* proof since this is weakly specified at the outset). They are:

SOURCE SPEC: $\forall e{:}nat\,\forall l{:}list \rightarrow \exists output{:}list$,

i.e., a weak specification for inserting an element, e, at the correct sorted position into a list, l, to obtain an output list;

TARGET SPEC: $\forall e{:}nat\,\forall a{:}nat\,\forall b{:}nat \rightarrow \exists output{:}list$,

i.e., a weak specification for inserting an element, e, at the correct sorted position into a list made up of $n = 2$ objects, a and b, of type nat, to obtain the output list.

$$\lambda a,\ \lambda b.\ (\lambda v2.\ (less(a, e, a :: v2, e :: a :: b))(\lambda v2.\ (less(b, e, b :: v2, e :: b))(nil)))$$

Figure 4–12: The extract for the specialized *insert* (sub)proof

For the sake of space we only reproduce the extract for the specialized *insert* proof (**fig.** 4–12). The average run-time, 26067.1 ms., over 100 samples, marks a 1.3 times improvement over the proof extract, **fig.** 4–11, where in the auxiliary *insert* function remains unspecialized.

The *insort* target program extracted from the proof wherein both the outer and inner (nested) induction schemas have been grounded can be envisaged simply by substituting the above extract for the $term\_of(insert)$ expression where ever it appears in the extract of **fig.** 4–10 (where the value, n, of the specialized parameter – in this case the length of the input list – will be the same for the *insert* sub-proof as it is in the main proof undergoing specialization).

4.2.4 Example 3: Normalization and Dependency Pruning: Automatically Specializing the *Upper Bound* Proof

To illustrate pruning we have reconstructed the specialization of Goad's main example algorithm, an *upper bound* algorithm, in the OYSTER environment.

Properties of PSS illustrated by *upper bound* Example

The main differences between this specialization and the previous ones are:

1. The source proof contains no induction schemata (i.e., no induction grounding will be involved). However, the source proof embodies a nested case split structure hence providing opportunities for *both* normalization and dependency pruning.

2. The source specification is complete in that it totally specifies the *upper bound* function. Furthermore, the target proof satisfies a specification which is identical to the partially evaluated source specification. Hence, the transformations from the partially evaluated source to the final (dependency pruned) target are source-to-target correctness guaranteed.

3. The specialization of the *upper bound* algorithm involves the instantiation of the *arguments* of the source proof (as opposed to, for example, the *length* of the input list).

The specification for the *upper bound* source proof is as follows:

SOURCE SPEC: $\forall x{:}nat\, \forall y{:}nat\, \exists z{:}nat.\ z \geq x + y \wedge z \geq x \times y.$

Following Goad, we shall abbreviate the above specification to $R(x, y, z)$, where the properties of R are expressed by the following lemmas (according to whether (1) $x \leq 1$, (2) $y \leq 1$, or (3) $\neg(x \leq 1) \wedge \neg(y \leq 1)$):

lemma 1 : $x \leq 1 \rightarrow R(x, y, y + 1)$;

lemma 2 : $y \leq 1 \wedge x > 1 \rightarrow R(x, y, x + 1)$; and

lemma 3 : $y > 1 \wedge x > 1 \rightarrow R(x, y, 2xy)$.

Initialization (Partial Evaluation)

Before any pruning can begin the proof must first be initialized. This is done more or less in the same way as the specialization of the *sumlist* proof: the mapping of specifications and of those branches corresponding to type checking are performed in an almost identical fashion except that rather than specifying a specific number of input objects in the target specification (corresponding to the length of the input) we instantiate a specific variable in the source specification with a specific value corresponding to the desired partial evaluation.

We assume that initialization consists of a specialization of $y = 0$, i.e., the variable y in the source specification (and proof) is instantiated to 0, yielding the following partially evaluated specification, which remains unchanged throughout the course of the subsequent pruning transformations:

partially evaluated target spec: $\forall x{:}nat\, \exists z{:}nat\, R(x, 0, z)$.

Pre- and Post-Conditions for Pruning[22]

Recall, §2.3, that since the OMTS rule-trees are akin to tactics (or proof-plans), that therefore the pruning transformations are akin to tactic (or proof-plan) transformation. Below we describe, both informally and formally, the conditions under which the pruning transformations apply, and the effects of their application on the proof (and program) construction.

- *Informal description*

The pruning transformations operate on case splits and the conditions under which they are applicable can be *informally* stated thus:

- *normalization* consists of removing all those branches in the source proof headed by a condition which evaluates to false under the chosen initialization; followed by

- *dependency pruning* consists of removing all those branches in the source proof which are associated with (non-identical) redundant computation. A proof branch (or sub-tree) is deemed redundant if an output for the main goal at the root node of the proof can be witnessed, and verified, without appealing to any of the facts established in that branch (or sub-tree).

Although dependency pruning need not necessarily be preceded by normalization, the pre-conditions required for dependency pruning a case split may be brought about by the redundancy being exposed through the normalization of a nested case split. This is because in normalizing the proof we remove a false condition, C_{false}, from the innermost instantiated case split, along with the corresponding

---

[22]The conditions apply to normalization and the subsequent dependency pruning, and not to the induction grounding transformations.

proof branch, and replace them by the true condition C_{true}. We shall call such a condition, produced as a post-condition of normalization, the *normalized condition.*

Following normalization, any further case split branches are then deemed redundant, and susceptible to dependency pruning, iff an output for the main goal can be established, and subsequently verified, without envoking any of the hypotheses, H, at the root node of that branch. Normalization can bring about this state of affairs if an output is witnessed and then subsequently verified by appealing to C_{true} *instead of* to H.

• *Formal description*

If we wish to establish some conclusion C through a proof by cases, by using the *decide* rule (§2.2.2), then this is tantamount to proving the sequent $A \vee B \vdash C$, where, as a general rule, $A = \neg(B)$. So normalization and dependency pruning can be seen as a pruning transformation for $\vee-$*elimination* proofs:[23]

Normalization corresponds to performing cut elimination on case analyses. If we let a lower case symbol, a, denote the instantiation of a term A such that a can be acertained either true or false (corresponding to the initialization, partial evaluation, stage of the specialization process) then normalization can be *formally* stated thus (using the refinement rule notational conventions of §2.2.2, and where $A \longmapsto B$ means "A transforms to B"):

$$\frac{P_{\phi_{H\vdash A\vee B}} \quad P_{\phi_{H,A\vdash C}} \quad P_{\phi_{H,B\vdash C}}}{P_{\phi_{A\vee B\vdash C}}} \longmapsto P_{\phi_{H,a\vdash C}} \quad if\ a\ is\ true,\ \text{otherwise}\ P_{\phi_{H,b\vdash C}}.$$

Dependency pruing can also be seen as a pruning transformation for $\vee-$ *elimination* proofs, and is defined *formally* thus:

---

[23]Both the pruning transformations can be shown to be guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. This is what Goad appeals to in order to render his specialization transformations correctness guaranteed. The PSS design circumvents the need for such "validity proofs" due to the presence of a target specification together with a target proof of that specification (*cf.* §4.3).

- $\dfrac{P_{\phi_{H\vdash A\vee B}} \quad P_{\phi_{H,A\vdash C}} \quad P_{\phi_{H,B\vdash C}}}{P_{\phi_{A\vee B\vdash C}}} \longmapsto P_{\phi_{H,A\vdash C}} \quad if\ A\ is\ discharged\ in\ P_{\phi_{H,A\vdash C}};$ [24]
- $\dfrac{P_{\phi_{H\vdash A\vee B}} \quad P_{\phi_{H,A\vdash C}} \quad P_{\phi_{H,B\vdash C}}}{P_{\phi_{A\vee B\vdash C}}} \longmapsto P_{\phi_{H,B\vdash C}} \quad if\ B\ is\ discharged\ in\ P_{\phi_{H,B\vdash C}}.$

So, in terms of the computation rule associated with an application of the *decide* rule (in order to prove the sequent $h : A \vee B \vdash C$), dependency pruning acts upon proof constructs of the following form:

$$decide(h, [P_{\phi_A}, \phi_{A\vdash C}], [P_{\phi_B}, \phi_{B\vdash C}]).$$

The effects (or post-conditions) of dependency pruning will be to replace the above with either the second or the third argument (precisely as in $\vee - elimination$). The pre-conditions which trigger such pruning depend on whether the assumption A (or B) was actually used – discharged – in constructing P_{ϕ_A} (or P_{ϕ_B}). Such information *cannot* be read from the extract terms, but is available for inspection within the proof P_{ϕ_A} (or P_{ϕ_B}), and within the corresponding rule-tree constructs.

This is clarified by example in **fig.** 4–13, wherein we have schematically represented the source proof for the *upper bound* program *before* initialization, where z is the *upper bound* for x and y in the universally quantified goal, and where we omit the initial $\forall - intro$ refinements. We have indicated which branches are subject to normalization, and the subsequent dependency pruning, when y is instantiated to 0.

Since the proof is syntactically rather dense, we have replaced the two sub-proofs, below the applications of *lemma* 1 and *lemma* 3 respectively, by dashed lines thus - - - - - -. These two "masked" sub-proofs are directly analogous to the displayed sub-proof below *lemma* 2 (except, of course, that they employ different lemmas), and all three sub-proofs collectively make up the verification component of the source proof. The reason we have chosen to show the sub-proof below *lemma* 2, in preference to either of the others, is because the corresponding case split branch (headed by condition $y \leq 1$) exhibits the required dependencies,

---

[24] Or, equivalently, $if\ A\ does\ not\ appear\ as\ an\ open\ assumption\ (free)\ in\ P_{\phi_{H,A\vdash C}}$.

between sub-goals and hypotheses, that enable dependency pruning (which is why we have labelled the hypotheses in this case branch by $h1, h2, ...etc.$).

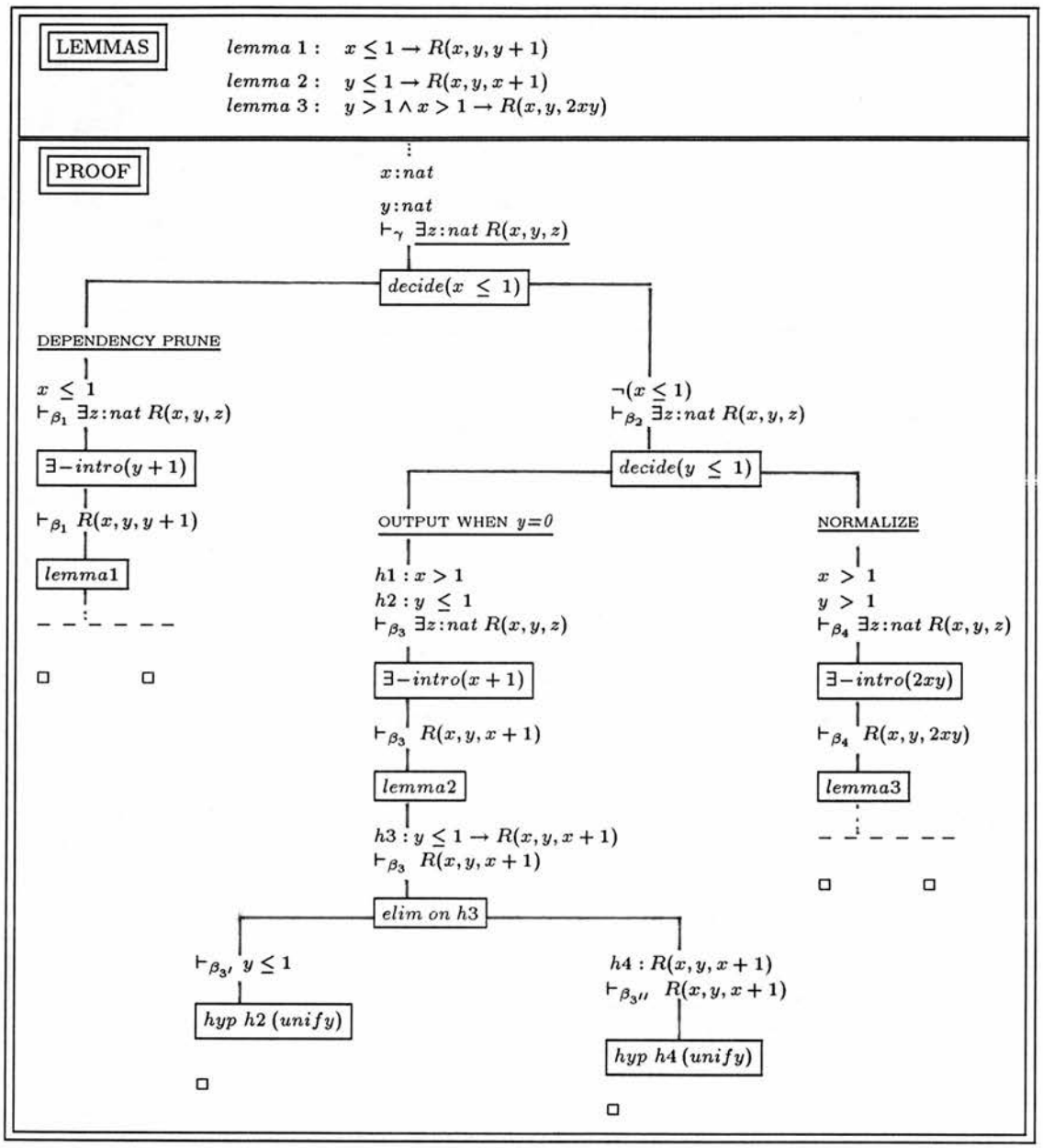


Figure 4–13: (Schematic) synthesis proof for *upper bound* program

The Rationale of Pruning (with Reference to fig. 4-13)

The branch which exploits *lemma 3* is pruned by normalization when the specialization is $y = 0$. This is because the innermost case condition, $y > 1$ is false when

$y = 0$. Hence, the right branch of the innermost case split is pruned and then replaced by sequencing in the true condition, $y \leq 1$, when $y = 0$.

The branch which exploits *lemma 1* is pruned by dependency pruning. This is because the proof branch which exploits *lemma 2* does *not* appeal to either case, $x \leq 1$ or $x > 1$, of the outermost case split in order to establish, constructively, that $x + 1$ is an upper bound for x and y. Instead, it appeals to a true hypothesis, $0 \leq 1$, that results from the instantiation of $h2 : y < 1$, and is sequenced in during normalization in place of the innermost case split.

So the complete passage from source to dependency pruned target relies on dependencies between the computational facts being available for exploitation. The extract program for the source proof is shown below in **fig.** 4–14.[25]

```
λ(x,
  λ(y,
    less(x, 1,                                          ;; if x < 1 then
     ((y + 1) ∧ term_of(lemma1) of x of y of axiom),    ;; upper bnd. = y + 1
    less(y, 1                                           ;; else, if y < 1 then
     ((x + 1) ∧ term_of(lemma2) of x of y of axiom),    ;; upper bnd. = x +1
     ((2 × x × y)                                       ;; else upper bnd. = 2xy
       ∧ (term_of(lemma3) of x of y of
             λ(_, axiom) of λ(_, axiom)))))))
```

Figure 4–14: The extract program for the *upper bound* source

The one on one relation between constructs in the extract and constructs in the proof from which it is extracted is self-evident, as is the fact that the relation is not bi-directional: dependency information in the proof, relating (sub)goals to hypotheses/assumptions, has no representative counterpart in the extract. This is precisely why proofs and the abstracted rule trees, but not programs, are subject

[25]Recall from *Chapter 2* that, regarding extract terms, _ signifies a bound variable in the extract term which is actually of no computational use. The term *axiom* signifies something which is axiomatically true, such as the existence of the type of natural numbers.

to dependency pruning. Since the relation between constructs in the rule-tree and constructs in the proof from which it is abstracted is bi-directional then dependency pruning can also be performed on the rule-trees.

In the following sections we describe, with the specific specialization of $y = 0$, the means by which the PSS performs the two pruning operations on the *upper bound* proof of **fig.** 4–13 (through transfrmations on the *upper bound* source proof rule-tree of **fig.** 4–14).

4.2.5 PSS Normalization pruning

In practice, normalization is performed, by the PSS, on the source proof *concurrently* with initialization (partial evaluation). Generally, after each operation of the pruning mechanism there will be some type-checking, syntax checking and, if necessary, correcting of both to make sure that OYSTER will accept any resulting (sub)proof.[26]

Once the PSS has abstracted the rule-tree representation of the source proof tree, then the argument value supplied by the desired partial evaluation, in our example $y = 0$, is substituted throughout the rule tree representation in place of the variable, y, which is being instantiated.

The new rule tree is applied to the mapped target specification. If during this application, any of the specialized proof case split conditions, which may now be evaluated, turn out to be false then the process does not continue in mapping the corresponding proof tree branch, i.e., that branch is pruned from the specialized proof tree. This corresponds to *normalization pruning.*

Regarding **fig.** 4–13, there are two case analyses for the *upper bound* OYSTER proof, corresponding to the following (case) splits:

---

[26]For example, if an integer, say 3, is substituted into the expression $x : int \Rightarrow y : int \Rightarrow z : int$ for y, OYSTER will *not* accept the resulting expression $x : int \Rightarrow 3 : int \Rightarrow z : int$ due to the term $3 : int$. Such a term can be removed from the expression without effecting the validity of the expression as a whole since the typing of integers need not be declared. This is all done automatically (the same applies to the dependency pruning).

SPLIT 1: $x \leq 1$ *or* $x > 1$; and

SPLIT 2: $y \leq 1$ *or* $y > 1$.

When the proof is partially evaluated (specialized) with y set to 0 then one case split condition becomes $0 \leq 1 \Rightarrow$ *void*. The proof branch corresponding to this case split condition can never be satisfied and is hence pruned from the tree. In place of the refinement $decide(y \leq 1)$ we *sequence in*, using the *seq* rule, the true condition $0 \leq 1$. This, then, is the *normalized condition*. As we shall elaborate in the next section, it is precisely this normalized condition which, in our example, enables the further dependency pruning to proceed.

The extract algorithm for the normalized proof is displayed in **fig.** 4–15. The

```
λ(x,
    less(x, 1,                                          ;; if x < 1 then
      ((0 + 1) ∧ term_of(lemma1) of x of 0 of axiom),   ;; upper bnd. = 0 + 1
      (λ(v1, (x + 1) ∧ term_of(lemma2) of x of 0 of v1), ;; upper bnd. = x + 1
            of λ(_, axiom))))
```

Figure 4–15: The extract program for the normalized *upper bound* proof.

removal of the λy term and the decision procedure $less(y, 1...)$ corresponds to the normalization of the source extract, where $v1$ is the normalized condition. It means we have one fewer function applications in the normalized extract than in the source extract. This causes the normalized proof extract to run 1.4 times faster than the source:

source *upper bound* **extract**: Average CPU time for 100 runs = 324.2 ms.

normalized *upper bound* **extract**: Average CPU time for 100 runs = 227.0 ms.

Alternative Normalizations

If we set the specialization such that the instantiated value for parameter y is greater than 1, say $y = 2$, then the PSS will normalize the inner-most case split by removing the left hand branch, along with the case conditions, and sequencing into the proof $2 > 1$ as the normalized condition.

Alternatively, if we had performed specialization on parameter x, say $x = 0$, then the PSS will normalize the outer-most case split by removing all the sub-proofs which depends on the condition $x > 1$, along with the case conditions, and sequencing into the proof $0 < 1$ as the normalized condition. Such a specialization, performed on a *non-nested* case split, will remove any opportunity for subsequent dependency pruning (hence limiting the practical usage of, normalization on non-nested case splits, to source proofs for which dependency pruning is not an option).

PSS Dependency pruning

As stated earlier, the rationale is simply that if a case split branch can be satisfied without appeal to any of the case split conditions then by retaining that branch we may remove everything else involved in the case analysis.

The specialization and normalization allow us to *dependency prune* the outer-most case-split, CASE 1, from the proof tree. Recall that in normalizing the proof we sequenced in the condition $0 < 1$ in order to replace the false condition, and the right hand branch, of the inner-most case analysis. This sequencing sets up two sub-goals: the first representing the original sub-goal with $0 < 1$ as an additional hypothesis, H_{NC}; and the second requiring us to establish that $0 < 1$. The latter is done simply through the application of simple arithmetical tactics, *arith*, followed by some type-checking, $wfftacs$, of the form 0 *in nat* and $0 < 1$ *in* $u1$.

Regarding the first sub-goal, the *normalized condition* is then appealed to in order to verify the witness for the output specified in the root node, *rather* than appealing to either of the case conditions, $x \leq 1$ or $x > 1$, of the outermost case split. Hence the case split is redundant and both the case conditions, along with the left hand branch, are dependency pruned.

A schematic representation of the target proof resulting from initializing, normalizing and then dependency pruning the schematic proof of **fig.** 4–13 is shown in **fig.** 4–16 below. The extract states that the upper-bound for inputs x and y,

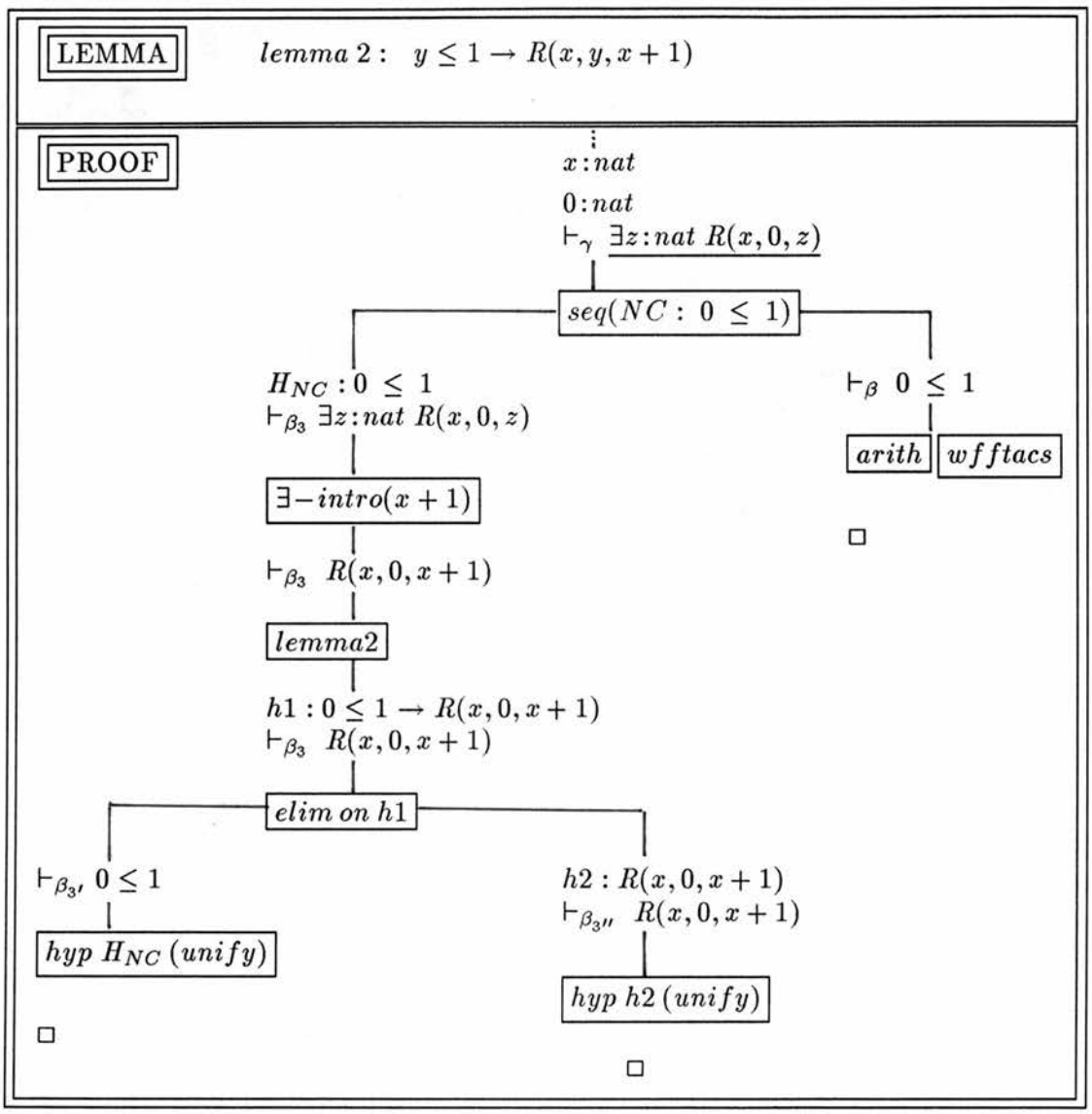


Figure 4–16: A schematic representation of the *upper bound* target proof.

where y is specialized to 0, is the upper bound provided by *lemma2*:

$$y \leq 1 \rightarrow R(x, y, x+1).$$

This is the case since the only branch incorporating one of the three lemmas and which remains after pruning is that branch incorporating *lemma2*.

Both the decision procedures, $less(x, 1)$ and $less(y, 1)$, have now been pruned, indirectly, from the initialized proof extract by performing normalization and dependency pruning respectively.[27]

The average cpu run time, over 100 runs, for each of the source, normalized, and dependency pruned *upper bound* proof extracts are displayed below.

Source *upper bound* **extract**: Average CPU time for 100 runs = 324.2 ms.

Normalized *upper bound* **extract**: Average CPU time for 100 runs = 227.0 ms.

Target *upper bound* **extract**: Average CPU time for 100 runs= 112.9 ms.

Summary of PSS Normalization and Dependency Pruning Methodology

In general, the specialization system can perform dependency pruning *automatically* by following procedures 1 - 7 below, the first three of which can be regarded as *preconditions* for dependency pruning:

---

1. Initialization, consisting of instantiating a proof parameter: the user indicates which variable, m, is to be assigned a desired value n.

2. Normalization, consisting of the removal of any case split, $C2$, for which one case condition, $cond2_{false}$, will always be false when (partially) evaluated with $m = n$. $cond2_{false}$, along with the case split, $C2$, are pruned and replaced by the true case condition, $cond2_{true}$. This condition, $cond2_{true}$, is the *normalized condition.*

3. Then, check to see if there is a further *outermost* case analyses, $C1$, with conditions $cond1_{left}$ and $cond1_{right}$. I.e., *before* normalization $C2$ must be dependent upon – nested within – one of the cases for $C1$.

---

[27]The $less(n, m)$ term represents the decision procedure corresponding to $n \leq m$.

4. *If yes*, can either of $C1$'s branches, $cond1_{left}$ or $cond1_{right}$, be satisfied by appealing to the normalized condition, $cond2_{true}$, sequenced in stage 2 above (normalization).

5. *If yes*, then we have a *candidate* for dependency pruning, i.e., all *pre-conditions* for dependency pruning have been met.

6. Failure or success then depends on whether or not either of $C1$'s branches can be satisfied by *not* appealing to either of its case analysis conditions, $cond1_{left}$ or $cond1_{right}$ at its root node.

7. *If yes*, then prune the *other* branch from the proof. For example, $cond1_{left}$ unifies with hypothesis $cond2_{true}$ and hence $cond1_{right}$, along with the case split $C1$, is redundant and therefore dependency pruned.

4.3 Advantages of the PSS Approach to Transformation

Below we summerize the advantageous, and in some cases novel, features of the PSS. In general, these can be seen to stem from the following basic properties of the PSS:

1. Transformations are performed on synthesis proofs of program specifications, and not the programs themselves.

2. This is achieved through tactic transformations on the rule-tree abstractions, which are akin to proof plans and contain:

   (a) all the information required to faithfully reproduce the *complete* proof from which it is abstracted, in particular the verification component of the proof; and

   (b) a record of the dependencies between facts in the proof.

3. At any stage of a specialization transformation (initialization, normalization, dependency pruning, or induction grounding) the system can, if the user desires, output a transformed proof which satisfies the specification embodied in the root node of the proof describing the transformed extract algorithm.

Transformations that Modify Input/Output Behaviour

If we schematically represent the normalization and dependency pruning specialization process thus:

$$u(x,y) \stackrel{initialize}{\Longrightarrow} u(x,0) \stackrel{prune\ case2}{\Longrightarrow} u(x,0) \stackrel{prune\ case1}{\Longrightarrow} u'(x,0) \quad where\ u \neq u',$$

and the following properties are true of the OYSTER system:

- $u \neq u'$;
- the proof, p', of $u'(x,y)$ is a transformation of the proof p of $u(x,y)$; and
- *both* u and u' satisfy the same (complete) specification,

then the transformation of input/output behaviour,

$$u(x,1) \stackrel{prune\ case1}{\Longrightarrow} u'(x,y),$$

is correctness guaranteed. This is a novel feature of the PSS since

- usual program transformations do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour; and
- Goad's system relies on proofs that (a) normalization is equivalence preserving, and (b) that dependency pruning will preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm (dependency pruning is *not* equivalence preserving).

The reason why, with this particular example, we are able to transform the functionality of an algorithm without altering the specification can *only* be because the specification is not complete, despite the fact that it fully specifies the upper bound function.[28] This point is not made by Goad, and it is easy to form the impression that a *completely* specified function can be altered without changing that specification. This is, of course, a contradiction in terms. In fact the *upper bound* specification is underspecified, albeit a full specification, in that through dependency pruning we can adapt the *range* of the inputs, from *all* naturals to a specific sub-set, without actually altering how we specified the input-output relation.

OYSTER Specification Language Provides Correctness Guarantee For Functionality Transformation

Recalling what was said in §4.1.3, Goad's system automatically performs dependency pruning on p-term abstractions, and then extracts a target program from the target p-term. This, in turn, means that the final target algorithm will have no corresponding proof and that therefore we lose:

(i) the correctness guarantee afforded by a target algorithm which is correct with respect to it's specification; and

(ii) the *source-to-target correctness* guarantee that both the normalized and dependency pruned extracts satisfy the *same* specification.

The fact that each stage of the PSS specialization process outputs a complete proof – through transformations performed on the rule-tree abstractions – means that we can, at each stage of the transformation, check to see that we still have a proof, albeit of *partial* status, which satisfies the specification. That is, individual sub-transformations do not lead to illegal proof steps in the target.

---

[28] *Cf.* §2.2.9, for a discussion concerning the distinction between full and complete specifications.

By virtue of the presence of a target specification and proof, we do not have to concern ourselfs with providing either equivalence proofs or proofs that certain transformations will preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. Furthermore, and again by virtue of the presence of a target specification, for those transformations, such as induction grounding, that are designed to alter a source program's functionality, by altering the specification and then propogateing the modification through the proof, we do not have to provide any separate correctness proofs for the target.

Automatability of the Pruning Transformations

It is largely due to the fact that dependency information, and thereby redundancy information, is rendered open for inspection, and modification, within the OYSTER proofs that accounts for why the OYSTER system can perform the pruning *automatically*: the lists of hypotheses and assumptions, including the case conditions, displayed at each node of a proof afford the pruning system with a *record of dependencies*, i.e., with a record of exactly what assumptions are appealed to in order to satisfy each (sub)goal. It then remains a mechanical task to determine which assumptions and hypotheses are redundant, and to prune the corresponding portions of the proof tree. Of course, not *all* of the assumptions in the proof hypothesis list will be relevant for pruning. Hence, when the specialization system constructs its rule-tree representation of the source proof it records, within rule-tree, *only* those assumptions corresponding to case split conditions. This generally cuts down on the search involved and the amount of information subject to manipulation (without effecting property 2(a) above).

Synthesis Proofs Enable (Automatic) Pruning

Conventional computational descriptions (such as functional recursive equations or some logic programming description) are *not* subject to the dependency pruning transformations: regarding the conditional *program* form of the *normalized* upper

bound algorithm,

$$u(x,0) \;=\; x \leq 1 \; then \; 1 \; else \; (x+1),$$

there is nothing that suggests that $u(x,0)$ can be *automatically* simplified, by the use of dependency pruning, to the expression $(x+1)$.

This is *not* the case with the PSS pruning mechanisms, in particular, the *inter-dependencies between the procedural commitments* made during the constructive synthesis are *explicitly* represented within the OYSTER proof, and within the PSS rule-tree abstractions. So for our example, the proof will contain a case analysis, CASE 1, whereby the case split is dependent on the size of x. Now, the fact that $(x+1)$ is an *upper bound* for both $(x+1)$ and $1 \times x$ does not depend on x being less than 1. This dependency information is contained in the OYSTER *constructive existence proof* and, via partial evaluation and pruning, allows for the *automatic* removal of the "computationally redundant" case split according to the size of x.

Extendability of the PSS

The availability of a complete proof at the termination point of the pruning transformations means that we have the choice as to whether to go on to perform further *proof* transformations. A practical example of this would be the specialization of a recursive program, by normalization and possibly dependency pruning, followed by the optimization of it's recursion schema (the latter forming the subject matter of *Chapter 5*).

Induction Grounding

The recursive behaviour of algorithms is specialized through induction grounding. This consists of partially evaluating and then pruning induction schemata in the proof describing the algorithm.

A Heuristic Setting

In addition to correctness preserving transformations, the system also has a *heuristic setting* that specializes algorithms extracted from proofs satisfying arbitrarily *weak* specifications. This allows us to place conditions upon the typing properties of all those programs that satisfy the weak specification.

4.3.1 A Brief Comparison with Bruynooghe *et al.*'s *EBL Based Transformation System*

Though similar to such systems as (Bruynooghe *et al.*, 1989; De Schreye & Bruynooghe, 1989), the PSS is an example of using partial evaluation in order to modify a program. However, in some respects it differs from the past transformation systems incorporating partial evaluation techniques: notably partial evaluation of proofs, followed by pruning, does not constitute a general optimization strategy solution, as is attributed to most EBL based systems. This is because partial evaluation, followed by pruning, is an example of *function transformation* and is designed to *adapt* input-output behaviour to a specific input (or a specific class of inputs), rather than simply *optimize* input-output behaviour.

The function of the proof pruning operations are, however, more sophisticated than those employed by, for example, (De Schreye & Bruynooghe, 1989). In the latter case a logic program is partially evaluated and then the associated execution tree is searched for repeated function calls – or *repeated sub-computations* – which are subsequently pruned. There is no additional information in the execution tree which suggests any non-repeated, but redundant, sub-computations.

In the case of program through proof specialization, once the proof has been partially evaluated then pruning is designed to remove redundancies corresponding to:

(i) *false computation* (i.e., sub-computations that evaluate to false);[29] and

[29]False computation would be exhibited through the partial evaluation of logic

(ii) *non-identical, but redundant, sub-computations.*

The reason that we can perform (ii) is that the objects of the transformation are *derivations* which, by their nature, contain dependency information. Since sub-goals along branches of a proof depend on the hypotheses, say $\{H\}$, at the parent nodes of the branches, then we can keep track of what branches are required for satisfying any particular end-node of the proof, say $\odot$, by observing which of $\{H\}$ are required in traversing alternative paths through the proof tree to $\odot$. In the case of an end-node within the scope of a partially evaluated case split, corresponding to $H \vee \neg H$, we can observe whether or not $\odot$ appeals to either of the hypotheses which constitute the case anaysis. If not then the case-split is not necessary and we can prune the branch which does *not* have $\odot$ as it's end-node.

4.4 Summary

In this chapter we have:

- described the PSS; and
- discussed the advantages of the PSS approach to specialization.

We have described:

- the specialization, or *grounding*, of inductive proofs as a means to transform recursive programs into non-recursive programs which perform the same task *for a specialized input*; and

---

programs and then observing the execution tree, although this is not discussed in (Bruynooghe *et al.*, 1989). Similarily, the removal of identical sub-computations through proof pruning would be a fairly trivial task: if distinct sub-trees of a proof are identical, save the variables appearing in formulae, then we can prune accordingly.

- the specialization of programs through the normalization and dependency pruning transformations performed on case analyses in the corresponding synthesis proofs.

The PSS satisfies the desirable criteria, for a transformation system, of correctness, generality and automatability.

In the case of pruning proofs with weak specifications, the source to target transformation can be treated heuristically although all transformations performed on the source are correct and the target proof is guaranteed to be well-founded.

One of the main differences between the PSS and previous transformation systems is that since proofs (or more accurately, proof-plans in the form of the rule-tree abstractions) are the objects of transformation, then information in addition to that concerned with simple execution can be exploited in the transformations.[30]

One big advantage of using OYSTER proofs as the objects of transformation, through the transformation of the abstracted rule-trees, is that *dependency information* is explicitly represented within the proof thus allowing alternatives (case analyses) to be pruned automatically if the partially evaluated proof exhibits redundancy. The non-identical, but redundant, sub-computations are located by the system automatically keeping track of which sub-computations depend on which hypotheses. If under a specific partial evaluation certain branches of case analyses need never appeal to *either* of the case conditions then the remaining branches may be pruned (i.e., clearly, a case split is not required).

So the *automatability* of the system derives from the presence of the dependency information, and the correctness system derives from the combined properties of the OYSTER proofs and the PSS rule-tree abstractions.

The main difference between the PSS and Goad's specialization system is that in the former rule-tree proof abstractions, or proof-plans, are subject to the pruning

---

[30]This feature being shared by the broader based transformation system, described in *Chapter 5*, of which the specialization system forms a sub-system.

operations (as opposed to p-calculus p-terms). This means that, in the specialization of *completely* specified algorithms, we end up with a target *proof* whose specification describes the desired target extract algorithm.

The presence of a (complete) program specification presents us with a novel state of affairs within the program transformation enterprise: by virtue of the presence of a target specification, we obtain a correctness guarantee for the source to target transformation of a program's functionality. Since Goad's system does not terminate with a distinct target specification, together with a target proof of that specification, then the pruning transformations must be limited to those that preserve the validity – but *not* necessarily the equivalence – of the source program with respect to the source specification. Furthermore, the fact that the PSS terminates with a target proof means that specialization can form the starting point of further *proof* transformations, i.e., we have the potential to "dove-tail" the specialization transformations with further proof transformation applications.

The nature of OYSTER proofs allows us to specialize the *typing* of the input/output relation specified in the source proof main goal. Hence, as in the case of the *sumlist*, *insertion sort* and the (nested) *insert* algorithms we can specialize an input of type *list* to an input consisting of a tuple of objects of type *int*. As such, the PSS transforms a source program's specification into a target specification that reflects the desired adaptation, whilst ensuring that the target program produced by the ensuing automatic specialization transformations is correct with respect to that specification.

Finally, as a precursor to *Chapter 5*, we note that both the PSS and the broader based optimization system (the OMTS) contain the following mechanisms which operate on the object-level proofs:

1. Procedures for representing proof trees in easily transformable list structures.

2. Mechanisms for *transforming* such proof representations (by *accessing, mapping and/or manipulating* sub-structures of the proof representations).

3. In particular, mechanisms for abstracting, and subsequently exploiting, dependency information from proofs.

4. Mechanisms for transforming, proof induction schema and, thereby, the recursion in the extract algorithm.

Chapter 5

Recursive Program Optimization Through Inductive Synthesis Proof Transformation

5.1 Introduction

In this chapter we describe the optimization of recursive programs through the automatic transformation of inductive (synthesis) proofs. We provide more detail, than given in *Chapter 2*, concerning the design of the inductive proof optimization sub-system, IPOS, of the MOPTS.

We discuss both the implemented strategies, and the design of some system extensions, for program through proof transformation. We provide examples and compare the systems performance with those reviewed in *Chapter 3*.

The chapter is separated into four main sections:

§5.2: We first discuss the transformation strategy of *tupling*, primarily within the context of recursive program transformation through the tupling transformations performed on inductive synthesis proofs. We keep the discussion at a fairly abstract level, leaving the details of how the IPOS performs the various analyses until §5.3.
Throughout **§5.2** we shall often have course to examine and discuss the properties of the source to target transformations, without at this stage concerning ourselves with the finer details of the role of the OYSTER proofs

themselves (this being left until §**5.3**). Toward this end, we shall abstract from the proof transformations a series of equation derivations which accurately reflect the *pattern* of re-writes, and in particular *unfoldings*, that are required during the construction of the target proof.

§**5.3:** Secondly, we provide details of the IPOS methodology for transforming proofs. We shall concentrate here on the role of the object-level OYSTER proofs, the implemented strategies and representation issues. In particular, we discuss how the IPOS represents and exploits information in the object-level source proof.
We provide a chart summarizing the role of the OYSTER proofs in the transformation strategy.
Thus far, we shall have concentrated primarily on source to target transformations wherein the induction schema (or rule) is transformed (§2.3.3). We conclude §5.3 by providing, firstly, an example of source to target transformations wherein the induction schema is retained but the induction cases are transformed. This takes the form of transforming nested inductions into single inductions. Secondly we provide a brief example of a source to target transformations which involves both the transformation of induction schemas and the transformation of induction cases.

§**5.4:** We discuss the merits of program through proof transformation, and make comparisons with those systems reviewed in *Chapter 3*. We shall concentrate especially on the automatability, correctness, generality and the control factors associated with the IPOS transformations.

§**5.5:** Finally, we include a description of how the system can be extended, within the same inductive proof transformation framework, to cope with a broader corpus of optimizations.

5.2 Using Tupling for Program Through Proof Transformations

We describe the transformation of recursive programs through the transformation of the dual induction schemata, where in the transformation is performed through the automatic transformation of one induction scheme to another.

We illustrate the methodology by describing example transformations, performed by the IPOS, of *course_of_values* recursion to *stepwise* recursion. Such transformations results in the optimization of an exponential recursive process to a linear recursive process and are achieved by:

- using *tupling* to construct a tuple object which groups together the separate recursive expressions in the source procedure;
- replacing the *course_of_values* induction employed in the source proof by a target *stepwise* induction; and
- using simple pattern matching to witness values for the tuple components at the base and step cases of the target schema.

In §5.5 we address, as further work, the transformation of a linear recursive process to a logarithmic recursive process. This falls within the same methodological framework as the aforementioned transformation, but should be regarded as an *extension* to the current implementation of the IPOS. The discussion of the linear to logarithmic transformation illustrates how successive optimizations of a program can be achieved by successively transforming the corresponding inductive synthesis proofs in accordance with the complexity of the computational rules associated with the various OYSTER induction schemas.[1]

[1]The reader may wish to glance ahead at **fig.** 5–11, §5.5, which lists the three inductions with which we shall be primarily concerned, along with the associated complexities.

5.2.1 Originality

Recall that the major focus of this thesis is an investigation of the potential of program transformation *through* constructive proof transformation. Apart from the *specialization* application, this high-level strategy is a novel one. In particular, the IPOS uses a novel means of optimizing recursive behaviour by combining transformation techniques with the transformation of source proof inductions.

The IPOS approach to program transformation has different ramifications concerning correctness, search and the use of dependency information. We briefly consider each of these in turn.

Correctness: As discussed in *Chapters 2* and *4*, the properties of the lower level OYSTER proof refinement system ensure that, given a *complete* source specification that once a target proof has been completed, then the meta-level transformation will be correctness guaranteed – simply because the target extract program satisfies the same complete specification as the source.

Reduced search space: The *proof* transformations allow us to remove the *fold* step from the unfold/fold strategy, §3.2.1, hence removing the associated control problems of deciding *when* to fold. The resulting strategy, consisting mainly of controlled unfolding, also has a reduced search space and, upon available evidence, is easier to automate.

Abstracting dependencies: By exploiting information abstracted from the proof – specifically the account of the dependencies between facts involved in the computation – the tupling transformations circumvent much of the computational analysis required by those systems, reviewed in *Chapter 3*, that employ the tupling technique ((Burstall & Darlington, 1977b; Darlington, 1981a; Chin, 1990)).

We shall return to each of these issues in §5.4.

5.2.2 Automatic Tuple Construction

To briefly re-cap on what was said, in *Chapter 3*, regarding the tupling technique, tupling is a *form* of *tabulation*, albeit constructed in real-time, since the tuple represents a record of previous recursive calls. The main advantage of tupling over the most general kind of table for redundant computation, *memo-tables* (Michie, 1968), is that we store only those subsidiary computations (or subsidiary *unfoldings*) required to make up the tuple.[2] It works by grouping together, in a single recursive tuple function, the separate recursive expressions in the source procedure. In the case of memo-tables there is a heavy storage requirement as entries inserted during function execution, are not usually removed even if they are no longer required.[3]

Conditions for Tupling

Recall from *Chapter 3* that, within the fold/unfold program transformation framework, (Chin, 1990) makes considerable advances in automating the tupling process in the functional programming language HOPE$^+$. The conditions for tupling, as originally specified in (Burstall & Darlington, 1977b) and (Pettorossi, 1984) and used in (Chin, 1990) for his tupling analysis, were stated in *Chapter 3* as:

1. there exist two or more *recursive calls* (or expressions), $f(n), ..., f(n-i)$, which share some *common recursion variable(s)* in a function definition (where $i \geq 2$), such that

2. we can construct a fixed sized tuple - the *eureka tuple* - within which common subsidiary recursive calls arising from the execution of each of $f(n), ..., f(n-$

---

[2]Or Eureka tuple as it is sometimes referred to in the literature, reflecting the difficulty in *automating* the tupling process.

[3]However, memo-tables do have the advantage of being more general in their range of function applications.

i) can be merged, thus forming a recursive function without the original redundancy.

Pre- and Post-Conditions for *Proof Tupling*

If we now return to *inductive proof transformation* we can restate the above conditions by appealing to the strong duality between recursion and induction (where we divide the conditions into *pre* and *post* conditions):

1'. *Pre-condition*: There exist two or more induction terms, $f'(n), ..., f'(n - i)$, which share some *common induction variable(s)* in a function definition (where $i \geq 2$).

2'. *Post-condition*: There must be present(constructed) a fixed sized tuple - the *eureka tuple* - within which common subsidiary function calls arising from the unfoldings of each of $f'(n), ..., f'(n - i)$ are merged, thus forming a recursive function without the original redundancy.

We shall refer to the tuple size, or the number of subsidiary calls tabulated within the tuple, as Φ.

Note that condition 1' is, in effect, a defining condition of *course_of_values* induction. This means that any proof employing one, or more, *course_of_values* induction schemes will generally be a good candidate for optimization by tupling.

However, functions which are constructed using schemas other than *course_of_values* induction can also satisfy condition 1' in an implicit sense. For example, a function, $f_{2\text{-}step}$, synthesized using $2-step$ stepwise induction may well be a candidate for proof tupling since an ivokation of $f_{2\text{-}step}(s(s(n)))$ will require *two* subsidiary calls on $f_{2\text{-}step}(s(n))$ and $f_{2\text{-}step}(n)$. We formally display the $2-step$ schema and provide an example of proof tupling on an instance of $f_{2\text{-}step}$ in §5.3.6.

A further class of candidate for proof tupling is auxiliary recursion. For example, if we have the following schematic definition

$$f(n) = f_1(n) + f_2(n - 1),$$

then it may be the case that upon unfolding either, or each of, f_1 and f_2, two or more induction terms, $f_j(n), ..., f_j(n-i)$, which share some common induction variable(s) are exhibited. This is the case with auxiliary recursive functions wherein the redundancy is not immediately obvious since it occurs amongst the auxiliary recursive calls (viz. the computation of the function(s), in the body of the definition, which are not self-recursive). Such "auxiliary redundancy" manifests itself in the source proof in the form of a *nested* induction. The task of proof tupling on such nested induction structures is to "merge" the computation associated with the innermost induction with that of the outermost induction.We shall address the optimization of these kinds of inductively synthesized functions in subsequent sections.

Henceforth, we shall distinguish proof transformations which employ a tupling technique from program tupling transformations by referring to the former as *proof tupling* and the latter as *program tupling.*

Analysing Definitions For Tuple Construction

The IPOS is concerned with the *real time* construction of *fixed sized tuples* (2' above). That is, tuples are constructed, of required fixed "size", Φ, during the course of transformation. Φ is determined by an analysis of the source proof definition - specifically what (Cohen, 1983) refers to as *descent functions.* Descent functions are those functions which are applied to the main recursive arguments used in subsidiary calls.

For example, consider the source *course_of_values* definition, fib_{cv}, for the *Fibonacci* function:

$$\begin{aligned} fib(0) &= 1; \\ fib(1) &= 1; \\ fib(n) &= fib(n-1) + fib(n-2). \end{aligned}$$

There are two subsidiary recursive calls entered in the source *course_of_values* proof in order to satisfy the induction step, $fib(n-1)$ and $fib(n-2)$. The corresponding two descent functions for the two subsidiary calls are in both cases

the *subtraction* function. We can determine Φ by a simple analysis of the descent functions: from the recursive step of the *Fibonacci* definition, we can see that in order to calculate $fib(x)$, for any x, we must "store", or tabulate, the subsidiary outputs for $fib(x-1)$ and $fib(x-2)$. The maximum difference between the recursive argument in the head and those of the subsidiary calls in the body is 2. Hence, in this case, Φ is 2.

Common Descent Function **Determine Φ**

The proof tupling process is suitable for linearizing what Cohen describes as the *common generator redundancy* class of programs. This class is represented by the below schematic definition for a function f, with two self-recursive calls, and where d_1 and d_2 are the descent functions:

$$f(x) \Leftarrow \textbf{if } b(x) \textbf{then } c(x) \\ \textbf{else } h(x, f(d_1(x)), f(d_2(x))).$$

The common generator redundancy class of programs are those programs where there exists a *common descent function*, δ, in terms of which d_1 and d_2 can be defined. This means each descent function is related to each other through δ in that each is cashed out in terms of applying δ a certain number of times, i.e., $d_1 = \delta^i$ and $d_2 = \delta^j$, where δ^i (δ^j) is to be interpreted as the application of δ i (j) times.

The general schematic function, shown above, for the bilinear common generator redundancy class of programs can hence be re-represented as $EQ.1$ below.[4]

$$EQ.1: \quad f(x) \Leftarrow \textbf{if } b(x) \textbf{ then } c(x) \\ \textbf{else } h(x, f(\delta^i(x)), f(\delta^j(x)))$$

So the *Fibonacci* recursive step can be represented thus:

$$fib(x) = fib((pred)^1(x)) + fib((pred)^2(x)),$$

[4]The common generator redundancy class also covers the class of programs, referred to by Cohen as the *Explicit Redundancy* class, where $d_1 = d_2$.

where *pred* is the predecessor function and takes the role of the common generator function δ (i.e., $\delta = pred$).[5]

By analysing the dependency graphs associated with common generator functions, Cohen discovered that they exhibit a common redundancy pattern which can be tabulated by using a table of size $max(i,j)$. Supposing that $max(i,j) = j$, the rationale is quite simply that in computing $f(x)$, where x is the recursion argument, the *maximum* number of potentially re-usable function calls will be equal to j since each of the j applications of the common function δ will correspond to a previous invocation of f (travelling down the recursion). Or alternatively, j will be the *maximum* number of times each subsidiary call of $f(x)$ will require dividing and conquering. Or alternatively again, j will be equivalent to the tuple size Φ.

So to evaluate $fib(x)$ *without* repeating any subsidiary function calls, we tabulate the *set* of all recursive calls stepping down from $fib(\delta^1(x))$ to $fib(\delta^\Phi(x))$. The tuple will include a parameter which acts as an accumulator, its value in successive invocations accumulating the value(s) of the function. This produces a *linear recursive* process, which is most naturally extracted from a proof employing *stepwise* induction. Hence the substitution of the target *stepwise* schema for the source *course_of_values* schema.

5.2.3 The Main Steps of the Proof Tupling

Recall that complete synthesis proofs have both a synthesis and a verification component (*chapters 2 and 5*). Similarly, we can compartmentalize the transformation of a complete source proof into a synthesis and a verification component such that the proof optimization process can be summarized as requiring the following steps:

1. SYNTHESIS:

---

[5]The IPOS is able to represent any destructor *or* constructor definition in this form, *cf.* §5.2.7 and §5.3.4.

(a) *Tuple Creation* (*Lemma generation*): The production of *both* an *explicit* and *recursive* definition of the target in terms of the source. By an *explicit* definition, we mean one that defines the tuple in terms of the subsidiary function calls that it tabulates, as opposed to a *recursive* definition that defines the target tuple in terms of itself.

(b) *Re-writing*: Using the source and (explicit) target definitions, perform numerous unfold applications on the recursive target definition.

(c) *Witness mapping*: Witnessing the existential quantifier at the recursive step of the target by matching and then mapping across (sub)structures from the source proof. These may undergo considerable transformation before being used as witnesses.

2. VERIFICATION: The instantiated induction cases are verified by appealing to source proof definitions and lemmas in order to guide the *unfolding* of the target recursive definitions until all (sub)goal terms match with target hypotheses.

To complete the synthesis component of the target proof requires witnessing a value for the recursive step of the target, i.e., (c) above. This involves:

- locating, and subsequently mapping, specific units in the IPOS rule-tree representation of the source proof;[6]

- performing substitutions consisting of replacing specific source constructs in the mapped units with constructs from the target proof so as to form a target unit; and

- entering the target unit into the rule-tree representation of the target proof.

---

[6]Recall, from *section 3.3, Chapter 2*, that rule-trees are skeleton proof representations which the MOPTS abstracts from OYSTER proofs.

We shall not concern ourselves with the details of this process until §5.3, and shall, instead, represent the witnessing stages of the transformation by a process involving the unification of meta-variables. We do, however, provide simple unification procedures for instantiating the meta-variables without specifying how, exactly, the IPOS achieves the desired unifications. This will enable us, in this section, to accurately represent *what* the goal of the transformations is – namely the completion of the witnessing stages followed by verification – without specifying *how* the various IPOS mapping and transformation operations accomplish the goal.

We shall first address the construction of an explicit definition for the target tuple. We shall then address the synthesis and verification components of the proof transformation process.

5.2.4 The Tuple Construction Procedures

Unlike the tupling transformations described in (Chin, 1990), the IPOS is designed with two alternative means of obtaining a suitable *explicit* definition of the target tuple. These are:

T1: a *descent function analysis procedure*, which by analysing descent functions appearing in the source proof induction step produces an *explicit* target tuple definition for a specified class of program (namely, programs that compute functions covered by $EQ.1$).

T2: a *heuristic tupling procedure*, which allows for the quick and efficient construction of an explicit target tuple definition, and is surprisingly successful over proofs which specify functions that satisfy the conditions for proof tupling (§5.2.2).

Under the current implementation, the specified class of program is somewhat limited. However, T1 can be regarded as a rational reconstruction, within the proofs as programs paradigm, of the program tupling strategy, and as such allows for a more direct comparison of *proof* transformation with *program* transformation. We show that the procedure will produce a target tuple for *both* that class of

program specified by *EQ*.1 *and* for a larger class of program. We shall also discuss how this larger class may be extended further (§5.5).

T1. Descent Function Analysis: *Common Descent Functions* Determine the Size of Tuple for *Common Generator* Programs

T1 constructs a suitable explicit definition for the target tuple by analysing the descent functions and their arguments, and thereby determining the degree to which each subsidiary recursive call is *divided* and subsequently *conquered.* This basically consists in determining $max(i,j)$, and then tabulating $max(i,j)$ subsidiary calls of the main function call in the head of the source definition recursive step.

Regarding the linearization of *Fibonacci*, if we let $fib_{tup}(x)$ be the auxiliary function used to define the tuple, the components of which take x as their data-structure, and interpreting $\langle fib(n), fib(m)\rangle$ as a function which constructs a tuple from the components of the *body* of the source equation, then this provides us with the following *eureka tuple*:

$$fib_{tup}(x) \;=\; \langle fib(\delta^1(x)), fib(\delta^\Phi(x))\rangle.$$

In the case of the *Fibonacci* definition, δ is the *predecessor* function and $\Phi = 2$, and so the above can be re-written as:[7]

$$fib_{tup}(x) \;=\; \langle fib(pred^1(x)), fib(pred^2(x))\rangle.$$

In general, if we schematically represent the recursive step of a bi-linear function, f, thus:[8]

$$h(x, f(\delta^i(x)), f(\delta^\Phi(x))),$$

---

[7] Or, equivalently, simply as:

$$fib_{tup}(x) \;=\; \langle fib(n-1), fib(n-2)\rangle.$$

[8] Note that we are not restricting i to 1.

then the *eureka tuple* for computing $f(x)$ is obtained by creating exactly Φ existentially quantified variables, where each quantifier ranges over one of the tuple components. Each quantified variable is then witnessed, respectively, by each member of the progression:

$$f(\delta^1(x)), f(\delta^2(x)), f(\delta^3(x)), ..., f(\delta^n(x)), ..., f(\delta^\Phi(x)),$$

to produce the tuple:

$$f_{tup}(x) = \langle f(\delta^1(x)), f(\delta^2(x)), f(\delta^3(x)), ..., f(\delta^n(x)), ..., f(\delta^\Phi(x)) \rangle,$$

where the value of the nth member will be a subsidiary call on f such that the value of its recursive argument is n less than that in the main call, such that, for example:[9]

$$f(\delta^n(x)) = f(x - n).$$

So, echoing Cohens analysis, the *common generator redundancy* class of programs can be characterized by recursive step definitions wherein the argument values of the recursive calls on f in the body differ in the number of applications of the predecessor function to the functions recursive argument. Such programs can be linearized by tabulating the members of the set of subsidiary calls between the main function call, $f(x)$, and that call in the body, $f(\delta^j(x))$, *where* $j = \Phi$, which takes the maximum number of applications of the *predecessor* function to the recursive argument x.

So the *EQ*.1, §5.2.2, characterization defines the class of programs whose recursive calls can be directly related in terms of the number of applications of the destructor function of f's data-structure, where the destructor function is the predecessor function.

---

[9]If we are dealing with constructor definitions, such that the common descent function, δ, is the successor function, s, then:

$$\underline{f(\delta^n(x)) = f(x + n)}$$

T2. A Heuristic for Tupling Functions

The simple *tuple formation heuristic*, T2, is capable of providing an explicit definition for the tuple for an *unspecified* class of functions. Although T1 *will* provide such a definition, as long as the source program is covered by $EQ.1$, T2 can provide the requisite tuples correctly in cases where the tupling procedure T1 would fail.

The tuple formation heuristic is simply to form the target tuple structure by a direct 1-1 mapping of the function calls in the body of the source definition recursive step. This is not, of course, guaranteed to produce the best tuple, but it will not produce a target program any less efficient than the source. The system will not produce an erroneous target program by employing T2, despite the fact that there are examples where an erroneous tuple would be produced by mapping the source recursive step (*cf.* §5.2.9). This is simply because the target specification, identical to that of the source, cannot be satisfied by a proof employing an erroneous tuple function.

However, for many simple self-recursive functions, the heuristic will provide the correct tuple. For example, for the linearization of the course_of_values *Fibonacci* program, we can simply map the two self-recursive calls in the body of the source step equation, thus providing a tuple of correct size, 2, and with the correct components, $fib(n-1)$ and $fib(n-2)$.

Similar considerations also apply to tupling where the source program computes an auxiliary recursive function such as the *factlist* function:

$$\begin{aligned} factlist(0) &= [\,]; \\ factlist(n) &= fact(n) :: factlist(n-1), \end{aligned}$$

where the auxiliary function $fact$ is defined as follows:

$$\begin{aligned} fact(0) &= 1; \\ fact(n) &= n \times fact(n-1). \end{aligned}$$

Here redundancy does not occur directly due to any self-recursive call but rather among the auxiliary recursive *fact* calls. This redundancy is exhibited by unfolding each subsidiary call in the recursive step once yielding:

$$factlist(n) = n \times fact(n-1) :: (fact(n-1) :: factlist(n-2)).$$

A one on one mapping of the auxiliary recursive, $fact(n+1)$ and self-recursive calls, $factlist(n)$, in the source recursive step equation will provide us with the correct explicit definition for the target tuple: $\langle fact(n+1), factlist(n)\rangle$. We shall return to this example in §5.2.8.

The Benefits of T2

For practical purposes, a lot less effort is expended by employing the "quick and dirty" heuristic method T2. In particular there is neither any descent function nor dependency analyses required to obtain an explicit definition for the tuple.

Furthermore, we do *not* loose the correctness guarantee of the source to target transformation: once the transformation terminates, then we have a target program which satisfies the target, and source, and a complete specification. So the correctness of the target extract program, with respect to it's specification, is independent of the means by which the target proof is constructed (i.e., by method T1 or by method T2). This fact highlights the advantages of *proof* transformation as opposed to program transformation.

In §5.2.8 we provide some further examples of source to target transformations by tupling. This will include implemented extensions to T1, specifically to deal with auxiliary recursive functions, and we shall, in §5.2.9, also present a couple of examples which show the limitations of T1 compared to T2, *and* vice-versa.

In §*5.3* we shall describe the *methodology* by which source proof constructs are mapped, transformed, and transferred to the target proof.

5.2.5 The Synthesis and Verification Components of the Proof Transformation

We can represent the synthesis and verification components of the proof transformation process by the sequence of equation developments presented below. For the present, we shall use meta-variables and interpret the goal of the synthesis component of the transformation process as the unification of these meta-variables in

order to witness a value for each tuple component without, at this stage, describing any of the matching and mapping operations that achieve the unification. We shall adopt the convention of using lower case for object-level symbols and upper case, of the form M_1, M_2...., for meta-level symbols.

The Synthesis Component

First the explicit definition for the target tuple is provided through either of the tuple construction procedures T1 and T2. Returning to the *Fibonacci* example, this produces an explicit definition for fib_{tup} in terms of the source definition:

$$fib_{tup}(n) \;=\; \langle fib(n+1), fib(n)\rangle.$$

Secondly, at the induction step of the target proof, we provide a definition for the recursive step of fib_{tup} in terms of the hypothesis (i.e., $fib_{tup}(n+1)$ in terms of $fib_{tup}(n)$):

$$fib_{tup}(n+1) \;=\; \langle M_1(u,v), M_2(u,v)\rangle, \;\; where \;\; \langle u,v\rangle \;=\; fib_{tup}(n).$$

The proof linearization then proceeds as follows, where M_1 and M_2 are the meta-variables:

$$\begin{aligned}
fib_{tup}(n+1) \quad &= \quad \langle M_1(u,v), M_2(u,v)\rangle, \; where \; \langle u,v\rangle \;=\; fib_{tup}(n);\\
\textbf{unfold } fib_{tup} \quad &\text{and} \quad \textbf{unfold } fib_{tup};\\
\langle fib(n+2), fib(n+1)\rangle \quad &= \quad \langle M_1(u,v), M_2(u,v)\rangle, \; where\\
&\qquad\qquad \langle u,v\rangle \;=\; \langle fib(n+1), fib(n)\rangle;\\
\textbf{unfold } fib \quad &\text{and} \quad \textbf{substitution } :u/fib(n+1)\&v/fib(n));\\
\langle (fib(n+1)+fib(n)), fib(n+1) \quad &= \quad \langle M_1(fib(n+1), fib(n)), M_2(fib(n+1), fib(n))\rangle.
\end{aligned}$$

We then use the source *proof* to provide values for M_1 and M_2. The first component of the r.h.s tuple (corresponding to the induction conclusion) results from *substituting* the *target induction hypothesis* tuple components for those in the *source induction step* in order to satisfy the first tuple component. Hence M_1 is instantiated to $+$. The second component results from a direct one on one mapping of the first component, $f(x)$, of the *target induction hypothesis.* Hence $M_2 \;=\; \lambda u, v \,.\, u.$

The Verification Component

The verification of the final equation above can then be represented as follows (where, in practice, the unfolding of occurrences of fib is done through the application of source proof lemmas):

$$
\begin{array}{rcl}
fib_{tup}(n+1) & = & \langle u+v, u\rangle \ \textit{where}\ \langle u, v\rangle = fib_{tup}(n); \\
\textbf{unfold}\ fib_{tup} & \text{and} & \textbf{unfold}\ \textit{using}\ fib_{tup}; \\
\langle fib(n+2), fib(n+1)\rangle & = & \langle u+v, u\rangle \ \textit{where}\ \langle u, v\rangle = \langle fib(n+1), fib(n)\rangle; \\
\textbf{unfold}\ fib & \text{and} & \textbf{substitution}\ :u/fib(n+1)\&v/fib(n)); \\
\langle (fib(n+1)+fib(n)), fib(n+1)\rangle & = & \langle (fib(n+1)+fib(n)), fib(n+1)\rangle.
\end{array}
$$

The control strategies incorporated within OYSTER, together with the particular form of derivation that OYSTER refinement proofs afford, mean, in effect, that *folding* is not a *necessary* requirement in order to introduce a recursion into the developing equations. This is because OYSTER inductive proof synthesis is driven by the heuristic requirement of matching induction hypothesis with induction conclusion, i.e., *fertilization*. This can be achieved purely by unfolding (or sequences thereof) both sides of the induction step until both head and body match or "merge". By repeatedly "unpacking" terms on both sides of the developing induction conclusion we eventually remove the induction term from the conclusion – the "unpacking", or *rippling out*, of the head(body) being facilitated by the current state of the body(head).

This form of proof development, which we shall refer to as *//-form*, has, we believe, advantages over the more traditional program derivations wherein rewriting is restricted to the body of the equations: most notably, that since the proof tupling transformations consist of unfolding target terms with specific source terms, then this removes the control problems associated with deciding when, and with what, to fold, thus reducing the search space (*cf.* §5.4.3).

The arguments, u and v, of the first component, the *"accumulator" component*, of the induction step tuple are local variables which act in a similar fashion to accumulators. The last argument, v, of the "accumulator" component is the desired output for the target algorithm. The remaining components – of which

there is only one, u, in the case of *Fibonacci* – serve as "records" of the subsidiary calls required to evaluate the accumulator component.

The Target Algorithm

We have already seen the λ-calculus extract term for the complete stepwise proof of *Fibonacci* (**fig.** 2–2, *Chapter 2*). Below we show a simplified version which illustrates more clearly that the call to the auxiliary fib_{tup} function appears as a subsidiary call of the main fib function call. This ensures that the target extract has the same functionality as the source extract. This directly reflects the way in which the target proof is constructed: in the course of transformation the IPOS enters, as a sub-goal, a new fact into the target proof stating the existence of a tuple with the required size Φ. This is done *after* the *Fibonacci* specification has had its universal quantifiers eliminated and *before* the target application of *stepwise* induction.

$$
\begin{array}{rcl}
n{:}nat\, fib(n) \Rightarrow nat; & & \\
\quad fib(n) & = & v \ \ where \ \langle u, v\rangle \ = \ fib_{tup}(n) \\
n{:}nat\, fib_{tup}(n) & \Rightarrow & \langle nat, nat\rangle \\
\quad fib_{tup}(0) & = & \langle 1, 0\rangle \\
\quad fib_{tup}(n+1) & = & \langle u+v, u\rangle \ \ where \ \langle u, v\rangle \ = \ fib_{tup}(n)
\end{array}
$$

5.2.6 Generality of the Proof Tupling

We are now in a position to generalize the tuple satisfaction procedures for *bi-linear* functions for which the recursive definition fits that of $EQ.1$, i.e.,

$$f(x) \ = \ h(x, f(\delta^i(x)), f(\delta^j(x))).$$

In order to satisfy, respectively, the first and second components of the tuple object introdued at the target step, the IPOS performs the following operations:

(i) substitute the i^{th} and j^{th} components of the **target induction hypothesis** tuple for those in the **source induction step.**[10]

(ii) map one on one the first i components of the **target induction hypothesis.**

Following these procedures results in a tuple, $f_{bi\_linear\_tup}$, schematically of the form:

$$f_{bi\_linear\_tup}(x) \;=\; \langle h(f(\delta^1(x)), ..., f(\delta^i(x)), f(\delta^{i+1}(x)), f(\delta^{i+2}(x)), ..., f(\delta^j(x)))\rangle.$$

Although this procedure will deal with tuples of any size, clearly carrying around very large tuples is computationally expensive and is an inherent limitation of tupling analysis. For programs requiring small size tuples, however, the efficiency gained by tabulating redundant calls in the source exponential program outweighs any additional processing involved in constructing the tuple.

General Procedure For //-Form Proof Transformation: the GPPT

By collecting together what has been said concerning the synthesis component of the transformation process, 1(a), 1(b), and 1(c) of §5.2.3, we now provide a general procedure, the GPPT, for removing redundancy from a source extract program, which executes an exponential procedure, by introducing tuples into the target proof. The procedure is correct only with respect to a specified class of source program (*cf.* $EQ.1'$ below). Further extensions are required to extend the procedure to accommodate, for example, auxiliary recursive functions (§5.2.8).

The GPPT is formulated such that it takes account of proof tupling which involves tuples of any size, depending on the value of Φ (where Φ is determined by the tuple construction procedures T1 or T2).

We shall use a more general schematic function representation, $EQ.1'$, than $EQ.1$ to represent the common generator redundancy class of programs:

---

[10]For bi-linear functions the j^{th} component of the **target induction hypothesis** will be the last component, and $j = \Phi$.

$EQ.1'$ $f(x) \Leftarrow$ **if** $b(x)$ **then** $c(x)$
else $h(x, f_1(\delta^{a_1}(x)), f_2(\delta^{a_2}(x)), f_3(\delta^{a_3}(x)), ..., f_k(\delta^j(x)))$

where the following holds:

- $j = \Phi$.
- x is the recursion variable (data-structure).
- The subscript k denotes the number of subsidiary calls in the source definition. Since we are dealing here with self recursive functions then it will always be the case that

  $$f_1 = f_2 = ... = f_k,$$

  and we therefore omit these subscripts in the subsequent discussion.
- As before, δ is the common descent function, and δ^{a_n} means a_n successive applications of δ.
- Each of $a_1, a_2, a_3, ..., a_k$ are the respective number of applications of δ to the subsidiary calls within the body of the source recursive definition, such that the following holds:

  $$a_1 < a_2,\ a_2 < a_3,\ ...,\ a_{k-1} < a_k$$

  Note that it is *not* necessarily the case that $a_n + 1 = a_{n+1}$, i.e., successive superscripts are not necessarily incremented by 1.
- We shall always assume that the subsidiary calls in the body of the source have been sequenced such that $f_1, f_2, f_3, ..., f_k$ take progressively *decreasing* argument values (i.e., the number of applications of δ in f_{k1} will either be the same or greater than that in f_{k2} as long as $k1 > k2$)

$EQ.1'$ extends the class of source definitions for the following reasons:

- It does not limit us to *bi-linear* functions, i.e., $k \geq 2$

- We need *not* assume that $k = \Phi$, i.e., $EQ.1'$ will allow for target definitions where the tuple size is not equal to the number of subsidiary calls in the source definition.[11]

- The respective recursive arguments of the subsidiary calls, $f_1, f_2, f_3, ..., f_k$ need not necessarily be only one application of the common generator function, δ, out of step. This is why arguments of the subsidiary calls in the body of $EQ.1'$ correspond to the progression

$$\delta^{a_1}(x), \delta^{a_2}(x), \delta^{a_3}(x), ..., \delta^{j}(x),$$

as opposed simply to

$$\delta^{i}(x), \delta^{i+1}(x), \delta^{i+2}(x), ..., \delta^{j}(x).$$

Examples

So, for example, the GPPT will optimize a tri-linear variant, *fib_tri*, of *Fibonacci* with a recursive step:

$$fib\_tri(n) = fib\_tri(n-2) + fib\_tri(n-4) + fib\_tri(n-6),$$

where the *tuple application function*, h, is $+$, and $i = 2$, $k = 3$, $\Phi = 6$, and the subsidiary calls are not simply one application of the common generator function, *pred*, out of step:

$$fib\_tri_0(n) = fib\_tri_1 pred^2(n) + fib\_tri_2 pred^4(n) + fib\_tri_3 pred^6(n).$$

Furthermore, h in $EQ.1'$ may be such that an actual instance of $EQ.1'$, say $fib\_tri_\times$ unpacks as follows:

$$fib\_tri_\times = (2 \times fib\_tri_\times(n-2)) + (7 \times fib\_tri_\times(n-3)) + (4 \times fib\_tri_\times(n-5)),$$

---

[11] This is the case, for example, with a variant of the *Fibonacci* function where the recursive step is

$$fib_3(n) = fib_3(n-1) + fib_3(n-3),$$

and hence, where $k = 2$ and $\Phi = 3$ (*cf.* §5.2.9).

where the subsidiary calls may be multiplied by some number. The tuple produced from such a definition would be:

$$\langle fib_{\times\_3\_tri}(n-1), (2 \times fib_{\times\_3\_tri}(n-2)), (7 \times fib_{\times\_3\_tri}(n-3)), \\ fib_{\times\_3\_tri}(n-4), (4 \times fib_{\times\_3\_tri}(n-5))\rangle.$$

Step by Step Description of the GPPT

The GPPT for linearization of exponential programs through proof tupling is provided in below.

- **Source Function Conditions**

1. Let f be an exponential algorithm synthesized through a course_of_values inductive proof, where:

$$f(n) \;=\; h(f(\delta^{a_1}(n)), f(\delta^{a_2}(n)), f(\delta^{a_3}(n)), ..., f(\delta^{j}(n))).$$

- **Tuple size, Φ, and Explicit Tuple Definition Determination**

2. Tuple construction procedure, T1 or T2, determines Φ and produces an explicit target definition for the new algorithm, f_{tup}:

$$(i)\;\; f_{tup}(n) \;=\; \langle f(\delta^1(n)), f(\delta^2(n)), f(\delta^3(n)), ..., f(\delta^{\Phi}(n))\rangle,$$

where $j \;=\; \Phi$

- **Tuple satisfaction procedures**

3. Letting $M_1, M_2, ..., M_\Phi$ be meta-functions (Φ many of them) then we can produce the following recursive definition for f_{tup}:

$$(ii)\;\; f_{tup}(inc(n)) \;= \\ \langle M_1(e_1, e_2, ..., e_\Phi), M_2(e_1, e_2, ..., e_\Phi), ..., M_\Phi(e_1, e_2, ..., e_\Phi)\rangle \\ where\;\; \langle e_1, e_2, ..., e_\Phi\rangle = f_{tup}(n),$$

where *inc* (increment) is the constructor function of its data-structure.

The source proof specification, along with the initial *introduction* rules, is mapped across to the target proof, and then (i) is entered into the target

proof, as a sub-goal, *before* the application of the target induction ensures that the linear target will have the same functionality – satisfy the same specification – as the exponential source extract.

Stepwise induction is applied to n, and (ii) is introduced at the induction step.

- **//-form unfold** (*steps 4 - 7*)

4. Set up the recursive 'definition' as a conjecture to prove.

5. Using the explicit definition of f_{tup}, (i), unfold the occurrences of f_{tup} on both sides of the equation.

6. Continue unfolding as follows:

   - *on the left hand side*: Using the step case recursive definition for f, (ii), unfold the first tuple element $f(\delta^i(inc(x)))$ (or unfold the last element, $f(\delta^\Phi(inc(x)))$, depending on how one constructs the data type); *and*
   - *on the right hand side*: Using the *where* clause, unfold each of $e_1, e_2, ..., e_\Phi$.

- **Unification of Meta-variables:**

7. *Unify the two sides of the equation*, instantiating each of $M_1, M_2, ..., M_\Phi$ (*see below*).

The Unification Step (*step 8*)

The GPPT, as specified above, leaves open the question of the unification of the meta-variables. Recall that, in practice, this unification is achieved through matching target structures required to witness a value for the output at the induction step, with the witnesses applied in the source proof induction. We shall provide further details in §5.3 since it is important to explain exactly how a *complete target proof* is obtained, thus providing the system with a correctness guarantee.

The unification sub-procedures will, in general, involve:

(i) Applying the function h to specific components in the induction hypothesis tuple, in order to obtain the accumulator component of the induction conclusion tuple.

(ii) Applying the *projection function*, ι, to each of the components in the induction hypothesis tuple, in order to obtain the other components of the induction conclusion tuple ($\Phi - 1$ many of them). Strictly speaking, the *projection function* takes two arguments, $\iota(n, l)$, where the first, n, is a natural number, and the second l, is a list of objects of any type. The output will then be the n^{th} member of l. In order to emphasize the role of the tuple size, Φ, we shall adopt a slightly different convention for representing the projection function, the list l will, in our analysis, always contain Φ elements (corresponding to the Φ tuple components), hence rather than using $\iota(n, l)$, we shall use $\iota(n, \Phi)$ to mean *the* n^{th} *member of a list of length* Φ.[12]

The procedure may also include incrementing the recursive argument of each induction hypothesis component by a constant factor.

Sub-procedures for unifying the meta-variables

By observing the source definition we can provide the following (sub)procedures, **A** and **B**, for the higher-order unification, *provided the source extract program computes a function covered by* $EQ.1$.

A and **B** constitute a more formal way of expressing the tuple satisfaction procedures, (i) and (ii), of §5.2.6, and, as such, should not mislead the reader when she or he comes to read §5.3, since they do indeed, at a fairly high level of abstraction, mirror the behaviour of the mapping and transformation operators of the IPOS.

---

[12]Since l may be large, this notation also serves as a convenient shorthand.

A: M_1 is the *tuple application function*, h in $EQ.1'$, applied to each of the $a_1^{th}, a_2^{th}, a_3^{th}, ..., j^{th}$ of M_1's arguments, i.e.,

$$M_1 = h(\iota(a_1, \Phi), \iota(a_2, \Phi), \iota(a_3, \Phi), ..., \iota(j, \Phi))$$

where ι is the projection function.

B: each of $M_2, M_3, ..., M_{\Phi-1}, M_\Phi$ are projection functions corresponding, respectively, to

$$\iota(1, \Phi), \iota(2, \Phi), ..., \iota((j-1), \Phi), \iota(j, \Phi)$$

5.2.7 Equivalent Tuple Representations Using Constructor or Destructor Functions

It is not important which calls are tabulated within the target tuple, but rather the relation between their respective arguments. So the important property of the explicit definition for the target tuple is that

(i) the difference between the recursive arguments in the body, and

(ii) the difference between each recursive argument in the body and that in the main function call in the head

remain the same. So, for example, the "constructor tuple" obtained from a constructor definition of *Fibonacci*, with recursive step $fib(x+2) = fib(x+1), fib(x)$, would be $\langle fib(x+1), fib(x)\rangle$. This differs from the "destructor tuple", $\langle fib(x-1), fib(x-2)\rangle$, obtained from the destructor definition of *Fibonacci*, with recursive step $fib(x) = fib(x-1), fib(x-2)$. However, both tuples are correct with respect to their source definitions since (i) and (ii) both apply in both cases.

In the case of the "constructor tuple", the common generator function is the *successor* function, s, such that $n+1 = s(n)$, and the *common generator* class of programs is re-charactarized by recursive step definitions wherein the argument values of the recursive calls on f in the body differ in the number of applications of the *successor* function to the functions recursive argument. So, regarding $EQ.1$, $\delta = s$ and $f_{tup} = \langle fib(s^1(x)), fib(s^0(x))\rangle$ such that $f(\delta^n(x)) = f(x+n)$.

5.2.8 Constructing an Explicit Definition for Auxiliary Functions

We now describe how the descent function analysis procedure, T1, is equipped to deal with evaluating Φ, and producing an explicit definition, for auxiliary recursive functions.

The situation for proof tupling *auxiliary recursive* functions is different. Common generator functions which are auxiliary recursive fit the following schematic definition $EQ.1''$:

$$\boxed{\begin{array}{l} EQ.1'' \quad f(x) \Leftarrow \textbf{if } b(x) \textbf{ then } k(x) \\ \qquad\qquad\qquad\qquad \textbf{else } h(x, f_1(\delta^i(x)), f_2(\delta^j(x))) \end{array}}$$

where $(f = f_1 \vee f = f_2) \wedge f_1 \neq f_2$. An example of such a function is the *factlist* function:

$factlist(0) = [\,]$
$factlist(n) = fact(n) :: factlist(n-1)$
where the auxiliary function $fact$ is defined as follows:
$fact(0) = 1$
$fact(n) = n \times fact(n-1)$

Here redundancy does not occur directly due to any self-recursive call but rather among the auxiliary recursive *fact* calls. This redundancy is exhibited by unfolding each subsidiary call in the recursive step once yielding:

$$factlist(n) = n \times fact(n-1) :: (fact(n-1) :: factlist(n-2))$$

where it is clear that each call on the main function $factlist(n)$ requires two subsidiary calls on the auxiliary function $fact(n-1)$.

The key to optimizing such functions is to combine, or *merge*, the computation of the auxiliary function call with that of the self-recursive call. In the case of $factlist$ we must, in effect, merge the recursion schema associated with the self-recursive $factlist$ call with that of the auxiliary recursive $fact$ call. In §5.3.6 we

shall see that the redundancy caused by the auxiliary function manifests itself in the form of a *nested* inductive sub-proof. The optimization is then performed, through proof tupling, by in effect merging the two inductions.

The tuple size, Φ, is determined by combining the tuple size, Φ_{aux}, that would be required for the auxiliary function considered on it's own, with the tuple size, Φ_{rec}, associated with the self-recursive call. So in the case of $factlist$, this produces a tuple size of 2, since from the recursive step of the $fact$ definition we determine by T1 that $\Phi_{aux} = 1$, and then by comparing the subsidiary factlist call in the body of the factlist recursive step with the main call we also determin, by T1, that $\Phi_{rec} = 1$. Hence $\Phi = \Phi_{aux} + \Phi_{rec} = 2$.

Similarly, the tuple size for the following variant, $factlist_2$, of $factlist$:

$$factlist(n) = fact(n) :: factlist(n-2)$$

would be 3, i.e. $\Phi_{aux} + \Phi_{rec} = 1 + 2 = 3$.[13]

Alternatively, if $factlist$ where defined thus:

$$factlist(n) = fact_2(n) :: factlist(n-2)$$

where $fact$ is defined through $2-step$ recursion thus:

$$\begin{aligned} fact(0) &= 1 \\ fact(s(0)) &= 1 \\ fact(s(s(n))) &= s(s(n)) \times fact(n) \end{aligned}$$

then $\Phi = \Phi_{aux} + \Phi_{rec} = 2 + 1 = 3$

The //-form Proof Linearization of $factlist$

Returning to the standard $factlist$ function, the //-form proof linearization proceeds as follows, where we have adopted the convention of placing a box around those terms to be unfolded ($factlist$ is abbreviated to $fctl$, and $fact$ to fct):

---

[13]Although this is not in fact the best tuple, *cf.* §5.2.9.

$$
\begin{array}{rcl}
\boxed{fctl_{tup}(n+1)} & = & \langle M_1(u,v), M_2(u,v)\rangle \ \textit{where}\ \langle u,v\rangle = \boxed{fctl_{tup}(n)}; \\
\textbf{unfold}\ fctl_{tup} & \text{and} & \textbf{unfold}\ fctl_{tup}; \\
\langle \boxed{fct(n+1)}, fctl(n)\rangle & = & \langle M_1(\boxed{u},\boxed{v}), M_2(\boxed{u},\boxed{v})\rangle, \ \textit{where}\ \langle u,v\rangle = \\
 & & \langle fct(n), fctl(n-1)\rangle; \\
\textbf{unfold}\ fct & \text{and} & \textbf{substitution}\ :u/fct(n)\&v/fctl(n-1); \\
\langle (n+1)\times fct(n), fctl(n)\rangle & = & \langle M_1(fct(n), fctl(n-1)), \\
 & & M_2(fct(n), fctl(n-1))\rangle.
\end{array}
$$

Finally, we must unify the two sides of the equation: the following procedures are adequate to supply the correct instantiations for simple bi-linear functions such as factlist.

A': M_Φ is instantiated to the dominant function, h, in the body of $EQ.1'$, applied to the *first* and *last* of M_Φ's arguments, i.e.,

$$M_\Phi = h(\iota 1\Phi, \iota\Phi\Phi).$$

B': $M_1, M_2, ..., M_{\Phi-1}$ correspond respectively to

$$h_2(inc(n), \iota 1\Phi), h_2(inc(n), \iota 2\Phi), ..., h_2(inc(n), \iota(\Phi-1)\Phi),$$

where h_2 is the dominant function in the body of the auxiliary function, e.g., in the case of $factlist$ the auxiliary function is $fact$ with recursive step $fact(n) = n \times fact(n-1)$. In this case h_2 is $\times$.

Hence, regarding the $factlist$ example, we instantiate M_Φ and M_1 as follows:

following **A'**:

$$M_\Phi \ = \ M_2 \ = \ h(\iota(1,\Phi), \iota(\Phi,\Phi)) \ = \ fact(n) :: factlist(n-1);$$

following **B'**:

$$M_1 \ = \ h_2(inc(n), \iota(1,\Phi)) \ = \ (n+1)\times fact(n).$$

5.2.9 Comparative Performance of the Tuple Construction Procedures

We now return to the initial production of an explicit definition, for the target tuple, in order to compare the performance of the tuple construction procedures T1 and T2.

T1 Versus T2

Clearly, the heuristic tuple construction procedure, T2, will only provide the requisite explicit tuple definition if the body of the source recursive equation contains exactly those subsidiary calls required to construct the tuple. This means, for example, that T2 will *not* find the required explicit definition for the variant, fib_3, of the *Fibonacci* function, where the recursive step of fib_3 is defined as follows:

$$fib_3(n) \;=\; fib_3(n-1) + fib_3(n-3).$$

T2 would erroneously produce the tuple $\langle fib_3(n-1), fib_3(n-3)\rangle$, whereas the requisite explicit definition for the target tuple, fib_{3_tup}, is:

$$fib_{3\_tup}(n) = \big\langle fib_3(n-1), fib_3(n-2), fib_3(n-3)\big\rangle.$$

This is correctly obtained by T1: the tuple size, Φ, is determined by $max(i,j)$, which in this example is $max(1,3) = 3$. We thus tabulate all 3 subsidiary calls $fib_3(n-1)$, $fib_3(n-2)$ and $fib_3(n-3)$ to obtain the correct explicit definition.

T2 Versus T1

There is a class of auxiliary recursive functions, the *race-ahead* auxiliary functions, for which the procedures for producing an explicit target tuple definition for auxiliary recursive functions will fail. For example, consider the *factlist* variant, $factlist_2$, of, where the recursive step of $factlist_2$ is defined thus:

$$factlist_2(n) \;=\; fact(n) :: factlist_2(n-2).$$

The tuple required to optimize the $factlist_2$ procedure is as follows:

$\langle fact(n), factlist_2(n-2)\rangle$.

This cannot, however, be determined in the same way as for the *factlist* function: by unfolding the subsidiary calls and then observing repeated calls to the auxiliary function *fact*. This is because the redundancy is only exhibited if we unfold each subsidiary call *twice*. This reveals that the call to $fact(n-2)$ is repeated. Two further levels of unfolding reveal that the call to $fact(n-4)$ is repeated, and so on.

The heuristic mapping procedure, T2, will provide the correct value for Φ, along with the required explicit definition: a direct mapping of the recursive step function calls, $fact(n)$ and $factlist_2(n-2)$, would achieve the correct, and in fact best, tuple $\langle fact(n), factlist_2(n-2)\rangle$.

5.3 The Proof Transformation Strategy of the IPOS

This section provides more detail than the overview given in *Chapter 2*, concerning the development of a target synthesis proof through the mapping, and subsequent transformation, of a complete source synthesis proof. We shall pay particular attention to the role of OYSTER proofs in the transformation, and how matching, and subsequent mapping and/or transformation, of source proof constructs achieves the witnessing stages of the target proof.

Recall from *Chapter 2* that the MOPTS abstracts a skeleton proof representation, the rule-tree, from the source, performs mappings and transformations on this, and then applies the transformed abstraction to the the source specification so as to attain a complete target proof. This process ensures that:

- the target extract is correct with respect to the target proof specification, which totally specifies the functionality of the target algorithm; and

- the functionality of the target is the same as the source, since they both satisfy the *same* specification.

The synthesis component of the transformation process is concerned with the formation of the target tuple, the replacement of the source induction by a suitable target induction schema and the subsequent witnessing of the target induction cases.

The verification component is concerned with evaluating the instantiated induction base case(s), and performing specific sequences of unfolding operations at the instantiated induction step using *both* source and target equations.

Both components involve:

(1) the fairly extensive mapping, and subsequent transformation, of constructs from the source proof; combined with

(2) heuristic theorem proving strategies (such as the ripple-out strategy discussed in *Chapter 5*); and

(3) transformation techniques such as *tupling.*

With regard to (1), by matching target sub-goals with source sub-goals, the IPOS determines to what extent it needs to patch the corresponding source proof branches in order to apply them successfully at the target sub-goals. For both the synthesis and verification components, we can conveniently categorize the proof constructs mapped and/or transformed from the source proof according to the purpose of the mapping and/or transformation as follows:

The synthesis component will involve mapping, and/or transforming, (sub) structures from the source in order to:

- obtain the target specification;

- evaluate Φ, and construct the target tuple;

- determine the nature, and number, of elimination rule applications; and perhaps most importantly,

- witness the existential quantifier at the target induction cases by mapping across structures from the source induction cases.

The verification component will involve mapping, and/or transforming, all those source proof branches associated with:

- tactics for controlling unfolding;

- well-formedness goals; (such as the applications of type-checking rules); and

- the application of lemmas – lemmas used for the satisfaction of the source induction cases are mapped across and, after some simple transformations, used by the unfolding tactics in order to satisfy matching target sub-goals.

5.3.1 The Common Design of the Proof Tupling and Specialization Optimizations

The transformation of a source proof, of a specification S, is depicted below, in **fig.** 5–1, along similar design lines as the specialization system, PSS (*cf.* **fig.** 4–1, *Chapter 4*).

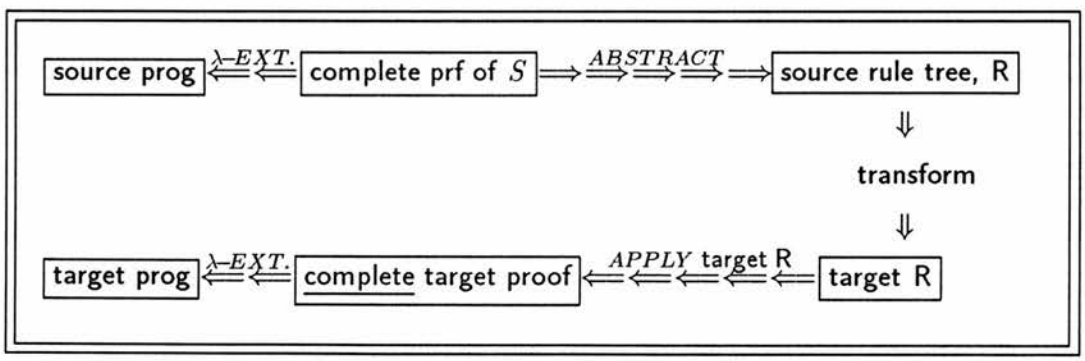


figure 5-1: The IPOS optimization process

The similarity between **fig.** 4–1, *Chapter 4*, and **fig.** 5–1 illustrates nicely the fact that the specialization and optimization transformations share the same system design.

5.3.2 A Graphic Example of Source to Target (Sub)-Structure Mappings

In *Chapter 2* we discussed at some length the object-level OYSTER synthesis of the source program for *Fibonacci*, using *course of values* induction, and of the target program, using *stepwise* induction. Using this as our standard example, the automatic IPOS transformation consists in replacing the source schema by the target schema, and then satisfying the *stepwise* cases using the tuple structure.

In **fig.** 5–3 we depict those (sub)structures of the source proof which are used to develop the target proof. Mappings from particular source proof branches to target branches are depicted by solid lines.

We also indicate, by dotted lines, the particular mappings utilized for constructing the target induction witnesses. The dashed lines indicate source proof constructs utilized for constructing the (sequenced in) target tuple.

5.3.3 The *Rule-Tree* Abstractions

Recall that in **fig.** 2–6 of *Chapter 2* we presented a simplified representation of the rule-tree abstracted from the *course_of_values Fibonacci* proof. In **fig.** 5–2 below, we present a more accurate representation of the rule-tree abstraction for the *course_of_values* proof for synthesizing *Fibonacci* from the complete specification:

$$\forall x{:}nat\, \exists y{:}nat\; fib(x) \;=\; y, a$$

and using the lemmas, 1 to 3, of **fig.** 5–3. The rule tree is akin both to a large

$(intro$ then $wfftacs)$ then $[elim(x, cv)$ then
$[decide(v2 = 0\ in\ nat)$ then
$[rewrite(v1)$ then $intro(s(0))$ then $wfftacs$ then $apply(fib\_lemma\ 1)$,
$decide(v2 = s(0)\ in\ nat)$ then
$[rewrite(v3)$ then $intro(s(0))$ then $wfftacs$ then $apply(fib\_lemma\ 2)$,
$(intro(v0\ of\ pred(v2) + v0\ of\ pred(pred(v2)))$ then
$wfftacs)$ then $apply(fib\_lemma\ 3)]]]]$

Figure 5–2: The source rule tree for *Fibonacci*.

OYSTER tactic, which combines a number of proof rules, and to a skeleton

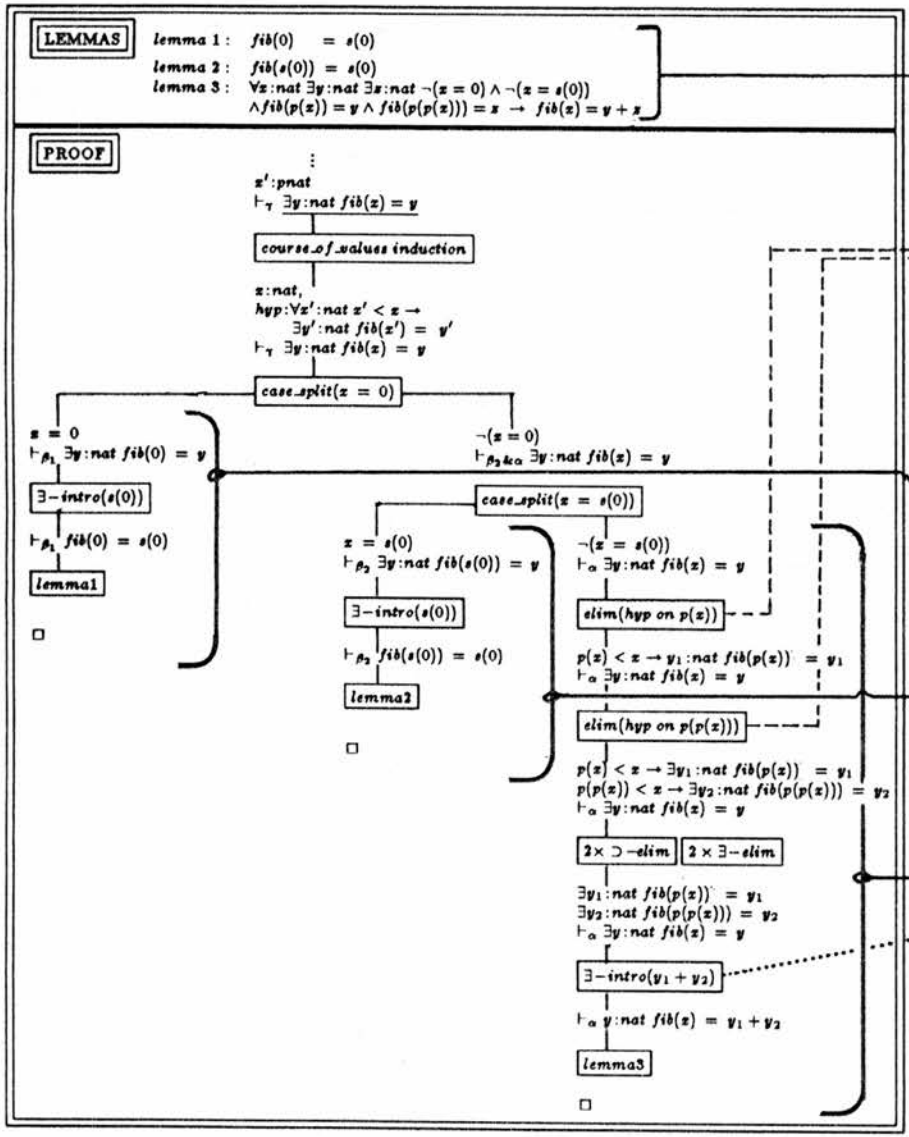


figure 5-3(a): Source Proof for Fibonacci using *Course_of_Values* Induction

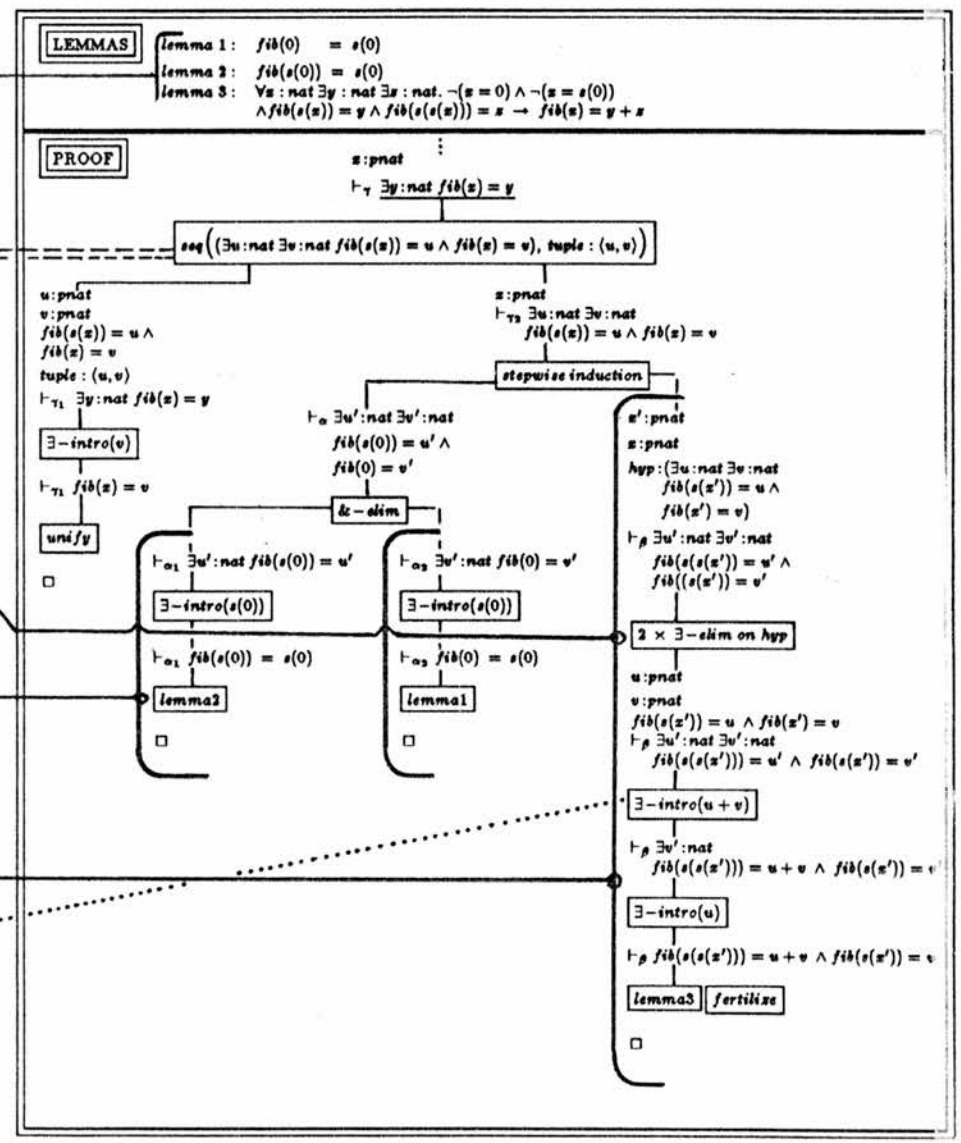


figure 5-3(b): Target Proof for Fibonacci using *Stepwise* Induction

Figure 5-3: Mappings between Source and Target *Fibonacci* Proofs

of a proof in which the inference rules of the proof, but not the formulae to which they are applied, are recorded.

Motivations for Employing Rule-Trees

The main motivations behind using the rule-tree abstractions as the objects of transformation, rather than the proofs themselves (or the OYSTER system's internal representations), is for similar reasons of efficiency and correctness that they are employed for the specialization transformations.

- *Efficiency*

Proofs will contain large amounts of information which is irrelevant to *both* execution and the tupling transformations. Hence inefficiency would result from this additional information being subject to extensive manipulation in the course of the transformations. This irrelevant information is not included in the rule-trees.

Conversely, the extract terms (or simply programs) have had much of the information relevant to the tupling transformations abstracted away, rendering any (automatic) optimization more problematic. An example of this is information required for determining Φ, concerning the dependencies between the main function call and subsidiary calls. This information is contained in the rule-tree abstractions, but not in the extract program. Hence tupling transformations that use programs directly must include some separate dependency graph construction and analyses. By using rule-trees we circumvent the need for any extraneous dependency graph construction and analyses.

As well as the formulae (goals) to which refinements are applied, further additional information contained in the proofs, but not in the rule-tree abstractions, is due to to the "unpacking" of large tactics, such that a single rule application that appears in the rule-tree may correspond to the application of a number of refinements in the proof that make up that rule. An example of this is, referring to **fig.** 5–2, the $rewrite(X)$ rule which, upon application, unpacks to a number

of refinements including substitution, ∀-elimination and various well-formedness refinements.

• *correctness*

Given the initial program specification,

- a rule-tree *contains all the information required to reproduce faithfully the* complete *proof from which it is abstracted, and*

- *at the termination point of a source rule-tree transformation, a target rule tree contains all the information required to produce the* complete *target proof.*

In this sense the rule-trees differ, for example, from Goad's Prawitz proof abstractions. The difference means that the target extract program is extracted from a complete target proof which, in turn, is automatically constructed through the application of the target rule-tree to the source specification. Hence the target program is *correct* with respect to its specification and with respect to the source specification.

• *Automation*

The rule-trees contain sufficient information to allow the source to target proof transformations to proceed *without* any user interference. In other words, in forming rule-trees from source proofs, the IPOS abstracts precisely that information which allows for the automatic construction of the target rule-tree.

The IPOS Exploitation of the Rule-Tree Structure

The nested structure of the rule-list reflects the branching pattern of the proof from which it was abstracted: the top portion of the source proof for the *Fibonacci* program involves the application of $\forall - introduction$, followed by an application

of course_of_values induction, followed by a couple of nested case splits. The case splits allow one to place conditions on the input such that we synthesize an output for *Fibonacci* depending on whether the input is 0, $s(0)$, or greater than $s(0)$.

Regarding **fig.** 5–2, the application of course_of_values induction on x is denoted by $elim(x, cv)$. Case analyses, other than that induced by the application of induction, are designated by *decide* expressions. There are two such nested *decide* applications represented within the rule-tree. The first of these corresponds to whether or not the induction candidate (renamed by OYSTER as $v0$) is 0. The first case, $v2 = 0$, is labelled by $v1$, and then $rewrite(v1)$ substitutes 0 for x in the main goal, and then $intro(s(0))$ witnesses a value for the output, y, of $s(0)$. The remainder of this proof branch consists of verification: by applying well-formedness rules, $wfftacs$, and by appealing to $fib\_lemma\ 1$ (**fig.** 5–3), we establish that if x is 0 then $fib(x) = s(0)$.

At the remaining case, we set up a further nested case split. Similar considerations as before apply to the first branch of the innermost case split, where $v2 = s(0)$. Here we must synthesize, and verify, an output for $fib(x)$ when $x = s(0)$: $rewrite(v3)$ substitutes $s(0)$ for x in the main goal, and then $intro(s(0))$ witnesses a value for y of $s(0)$. The output is verified by applying well-formedness rules, $wfftacs$, and appealing to $fib\_lemma\ 2$.

The remaining branch of the innermost case split corresponds to the inductive step of the proof where $v2 > s(0)$. This fact need not be stated since it follows directly from the other case conditions. The rule $intro(v0\ of\ pred(v2) + v0\ of\ pred(pred(v2)))$ witnesses a value for the induction step by substituting $pred(v2)$ and then $pred(pred(v2))$ for the recursive argument in the induction hypothesis $v0$. As is the norm with course_of_values induction this establishes that $pred(v2)$ and $pred(pred(v2))$ are less than $v2$, and that therefore there is an output for each (denoted by $v0\ of\ pred(v2) + v0$ and $v0\ of\ pred(pred(v2))$). Finally the induction step is verified by applying $wfftacs$ to satisfy well-formedness goals, and then appealing to $fib\_lemma\ 3$.

The reader should note that, as for the specialization process (*Chapter 4*), an account of the dependencies between facts involved in the computation can be ab-

stracted from the source rule-tree proof representations for exploitation during the transformation process. For example, to witness values for the base cases we must appeal to the case split conditions, $v1$ and $v3$, and to witness a value for the induction step we appeal, twice, to the induction hypothesis $v0$. Regarding verification, the rule-tree informs us that we must appeal to specific lemmas: $fib\_lemma\,1$ and $fib\_lemma\,2$ for the base cases, and $fib\_lemma\,3$ for the step case.

5.3.4 Exploiting Dependency Information for the Target Rule Tree Construction

We now provide some further detail concerning the exploitation of source proof dependency information in order to construct the target rule-tree: in particular the sub-list which, upon application, is responsible for witnessing the target stepwise induction cases, and for subsequently verifying the case instantiations.

Regarding **fig.** 5–3, if one looks at the proof branch corresponding to the step case of the course_of_values induction then it is clear from the proof node containing the following hypotheses and goal:[14]

$$
\begin{array}{lll}
\underline{hypotheses}: & x-1<x \;\rightarrow\; y_1{:}nat\; fib(x-1) = y_1 \\
& x-2<x \;\rightarrow\; y_2{:}nat\; fib(x-2) = y_2 \\
\\
\underline{goal}: & \vdash\; \exists y{:}nat.\; fib(x) = y \\
\\
\underline{refinement}: & \boxed{\exists{-}intro(y_1+y_2)}
\end{array}
$$

that, for any x, in order to construct an output for $fib(x)$, we must appeal to both the subsidiary calls $fib(x-1)$ and $fib(x-2)$, through unfolding with the two hypothees.

---

[14]Where convenient, we shall use $x-1$ and $p(x)$ interchangeably (similarly for $x+1$ and $s(x)$).

Since the maximum difference between the recursive argument in the induction step goal, $\exists y : natfib(x) = y$, and those of the subsidiary calls, $x - 1$ and $x - 2$, used to witness a value for y is 2, then the required tuple size is 2.

Furthermore, we know from the witnessing step, $\exists - intro(y_1 + y_2)$, which unfolds to the following:

$$\exists - intro(fib(x - 1) + fib(x - 2)),$$

that a value for $fib(x)$ is obtained by adding together those two subsidiary calls which take recursive arguments that differ from x by a value of 1 and 2 respectively.

Hence, we have all the information needed for the target tuple construction: we require a new fact to be entered into the target proof stating the existence of a tuple of two components (i.e., $\Phi = 2$). Such a fact can be entered into the target proof, as a new sub-goal, by a generalized version of the *sequence*, or *seq*, rule (§2.2.2).

In our example, the sequenced fact is:

$$seq(\{u : nat \,\&\, (fib(s(x)) = u)\} \# \{v : nat \,\&\, (fib(x) = v)\}.$$

This is entered into the developing target rule tree, and upon application will produce the corresponding target sub-goals (*cf.* **fig.** *5–3*).

Stepwise induction is applied at the second subgoal in order to prove the above fact. This is represented by the term $elim(x, new[ind\_obj, step\_hyp])$, within the target rule-tree, where *ind_obj* is the induction variable and *step_hyp* the induction hypothesis.

At the base case sub-goal an *intro* rule is applied which, in this context, has the effect of decomposing the goal into the separate tuple components. Such decomposition of the tuple will always be controlled by the tuple size, Φ. We then map across the base case witnesses, 0 and $s(0)$, from the source proof in order to witness a base case value for each of the tuple constituents u and v. The base cases are then verified by mapping across and applying the source base case lemmas. So the target rule-tree unit responsible for the induction base case will be of the

structure depicted in **fig.** 5–4, where the first *intro* rule splits the base case goal into two, and $wfftacs$ tactically applies various well-formedness rules.

$[intro]then$
$\quad [intro(s(0))then$
$\quad\quad [wfftacs, fib\_lemma\ 1]$
$\quad intro(s(0))then$
$\quad\quad [wfftacs, fib\_lemma\ 2]],...$

Figure 5–4: The base case target rule-tree construct

At the stepwise induction step, we must then witness a value for $fib(x+2)$, the first tuple component, by appealing to those two subsidiary calls which take recursive arguments that differ from $x+2$ by a value of 1 and 2 respectively, namely: $x+1$ and x. We also know, from the witness at the induction step of the source proof, that the main function call requires *adding* the Φ subsidiary calls. So the *first* target tuple component, $fib(x+2)$, takes as a witness $fib(x+1)+fib(x)$. In other words, the system substitutes the subsidiary calls in the target induction hypothesis for those in the source induction conclusion. Such substitutions are the means by which the IPOS realizes the tuple satisfaction procedures (which are stated both in English, as (i) and (ii) of §5.2.6, and formally, as **A** and **B** of §5.2.6).

The instantiated induction step sub-goal is then verified by appealing to the same tactics for unfolding *and* the same lemma, $fib\_lemma\ 3$, as used to verify the source induction step.

A similar analysis of the source proof could be performed to obtain the other, second, component of the target tuple. However, this will always be provided by one of the target hypotheses and can hence be directly appealed to in order to witness a value for the remaining component.

Regarding the use of lemmas, the IPOS is equipped with a simple translation procedure that turns a destructor type lemma of the form:

$$f_1(x) \;=\; f_2(f_1(x-a), f_1(x-b)), \;\; where\; b \geq a,$$

into a constructor version of the following form:

$$f_1(x+b) \;=\; f_2(f_1(x+(b-a)), f_1(x)).$$

Hence there is no problem in using source proof lemmas that define a function $f(x)$ in terms of *predecessors* of x, since, if necessary, we can translate it into the equivalent lemma that defines $f(x)$ in terms of *successors* of x.

```
intro then
  [seq({u : nat & (fib(s(x)) = u)} # {v : nat & (fib(x) = v)}, new[tuple]) then
    [elim(x, new[ind_obj, step_hyp]) then
        [intro then
            [intro(s(0)) then
                [wfftacs, apply_lemma(fib_lemma 1)],
             intro(s(0)) then
                [wfftacs, apply_lemma(fib_lemma 2)]],
         elim(step_hyp, new[a3, a4, a5]) then
            [& - elim then
                [intro(plus(a3, a4)) then
                    [wfftacs, apply_lemma(fib_lemma 3),],
                 hyp(a3)]]],
         elim(tuple, new[a6, a7, a8]) then [hyp(a7)]],
wfftacs]
```

Figure 5–5: The target rule-tree for *Fibonacci*

The complete target rule-tree, including *both* the induction cases, will have the structure displayed in **fig.** 5–5. The key rule applications, resulting from applying the target rule-tree of **fig.** 5–5, have the following effects:

- The $seq(\{u : nat \;\&\; (fib(s(x)) = u)\} \# \{v : nat \;\&\; (fib(x) = v)\}, new[tuple])$ rule introduces a new node in the proof tree with two subnodes where one, $G1$, represents the original subtree with an additional hypothesis and the other subnode, $G2$, is responsible for proving the hypothesis.

$G1$:

- The $elim(x, new[ind\_obj, step\_hyp])$ is the application of *stepwise* induction on x (the same induction candidate as in the source), and where *ind_obj* is the induction variable, and *step_hyp* the stepwise induction hypothesis.

- The commands $intro(s(0))$ and $intro(s(0))$ are the witnessing steps of the two base cases (i.e., outputs for $fib(0)$ and $fib(s(0))$ respectively). The instantiated base cases are then verified by the lemmas $fib\_lemma\ 1$ and $fib\_lemma\ 2$. *All* this information is mapped across from the source rule-tree.

- The $elim(step\_hyp, new[a3, a4, a5])$ rule strips both the existential quantifiers from the induction hypothesis and renames the bound variables such that $a3 = u = fib(s(x))$, $a4 = v = fib(x)$, and $a5 = \langle a3, a4 \rangle$.

- The $intro(plus(a3, a4))$ witnesses a step case value for the first tuple component $fib(s(s(x)))$. This is obtained by substituting $a3$ and $a4$ for the arguments in the source induction step.

- The $apply\_lemma(fib\_lemma\,3)$ command is mapped across from the source proof in order to verify the target induction step.

- The $wfftacs$ tactic application is also mapped from the source in order to satisfy the well-formedness goals.

- The $hyp(a3)$ command is used to witness a value for the remaining, second, tuple component, $fib(s(x))$ by appealing to the induction hypothesis $a3$.

$G2$:

- The $elim(tuple, new[a6, a7, a8])$ rule decomposes the tuple, synthesized via $G1$, into its constituents: $a6 = u = fib(s(x))$, $a7 = v = fib(x)$, and $a8 = \langle a3, a4 \rangle$.

- The rule application $hyp(a7)$ provides an output, y, for *Fibonacci* by appealing to hypothesis $a7$.

We can have a good expectation that the same pattern of unfolding employed in the source proof will also succeed in the target proof since this is the general *ripple-out* strategy that is shared by the majority of inductive proofs (§2.2.4): as with the CLAM automatic proof-planner the ability to provide typical proof strategies, such as the rippling out stages of a proof verification, assists with the automatic completion of the target proofs.

5.3.5 Summary of the IPOS Proof Tupling: A Strategic Plan

We now provide a transformation plan which summarizes program optimization by the transformation of inductive synthesis proofs, and which pays particular attention to the role of the OYSTER proofs and the rule-tree abstractions. We shall have course to refer to example source to target mappings and shall use the Fibonacci example, depicted in **fig.** 5–3. Consequently, examples of source rule-trees will refer to **fig.** 5–2. Text appearing within a frame-box describes how the rule-trees relate to each step of the transformation. For ease of explanation, we sometimes refer to the direct application of a target rule *before* the completion of the target rule-tree. In practice, no target rules are applied until the transformation terminates with a complete target rule tree. The rule-tree is then applied, rather like a large tactic, to the specification.

---

1. Abstract a rule-tree representation, R_s, from the source.

The structure of proofs is preserved in the nested list structure of the rule-tree abstractions. Any sub-list within the rule-tree is headed by a representation of a proof node, and the subsequent nested sub-lists collectively represent the sub-proof below that node (each of which can be automatically located as specific sub-lists within R_s). The rule-tree is, in effect, a skeleton of a proof in which the inference rules of the proof, but not the formulae to which they are applied, are recorded. Open assumptions in a proof are mapped to free variables in the abstracted rule-tree, and discharges of assumptions correspond to bindings of variables. A complete proof is obtained upon applying the rule-tree to the specification of the proof from which it was abstracted.

2. Form a new target rule tree, R_t, at this stage an empty list.

3. Map one on one the target specification from source specification, thus providing common source and target specification, S:

$$\forall x{:}nat.\ \exists y{:}nat.\ fib(x) = y,$$

where x and y are, respectively, the specified input and output.

S is *not* entered into R_t which, as with all rule-trees, does not represent the formulae to which refinements are applied. R_t will be *applied to S once it is completed to form a complete target proof with the same functionality as the source.*

START OF R_t CONSTRUCTION

4. Locate and map across the initial source proof elimination rules.

These will correspond to the outer-most sub-lists headed by an `intro` rule application.

5. Locate the unit (sub-list), R_{ind}, within R_s corresponding to the source application of induction and the subsequent sub-proof tree.

R_{ind} will correspond to the nested sub-list structure headed by an `elim(X,Ind)`, where X is the induction candidate and Ind the chosen induction schema. In our example R_{ind} will be headed by `elim(x,cv)`, i.e., the application of course_of_values induction to x.

6. Locate within R_{ind} and form a record of the induction variable, the source induction base case(s), SB, and the induction conclusion, SC.
In our example these correspond to the following:

   - *induction variable*: x;

   and the case conditions and corresponding witnesses for the

   - *source induction base case(s)* (SB): $x = 0 \rightarrow fib(x) = s(0)$ and $x = s(0) \rightarrow fib(x) = s(0)$;
   - *induction conclusion* (SC): $fib(x) = fib(p(x)) + fib(p(p(x)))$.

Case splits used to partition the base case inputs correspond to the nested sub-list structure headed by a `decide(C)` rule application, where `C` is some case condition. The witnesses will be located as sub-lists within R_{ind}, headed by an `intro(W)` rule application, where `W` is an object of the same type as the output for S. In the step case for our example the witness will be of the form `intro(v0 of pred(v2)+v0 of pred(pred(v2)))`, where `v0 of pred(v2)` means the evaluation of the induction hypothesis, `v0`, when x is set to the predecessor of the induction variable `v2`.

7. Using either of the target tuple construction procedures, T1 or T2, produce a value for the tuple size, Φ, and, thereby, produce an explicit definition, $tuple_{def}$, for the target tuple stating the existence of a tuple which tabulates Φ components, where each component, $t_1, t_2, ..., t_\Phi$, is of identical type to the *induction variable*, SI (or, alternatively, the consequent of S); t_1 will contain the output Y.
   In our example, $tuple_{def}$ corresponds to

$$\exists u{:}nat.\ \exists v{:}nat.\ fib(s(x)) = u \wedge fib(x) = v,\ tuple{:}\ \langle u, v\rangle,$$

   where v contains the *Fibonacci* output.

No additional dependency graph construction, nor subsequent analyses, is required to evaluate Φ and, thereby, to produce $tuple_{def}$. The required information is read directly from the witnessing unit, `intro(v0 of pred(v2)+v0 of pred(pred(v2)))`, within R_{ind}.

8. Using the generalized version of the *seq* rule a new fact is entered into the target rule-tree R_t. This has the desired effect of introducing a new node in the proof tree with two subnodes $G1$ and $G2$, where $G1$ represents the original subtree with an additional hypothesis stating $tuple_{def}$, and where $G2$ is responsible for proving $tuple_{def}$.

   SYNTHESIS COMPONENT OF $G2$

9. Apply the *stepwise induction* tactic at $G2$, choosing the same induction variable as used in the source *course_of_values* induction.

10. The *stepwise induction* tactic sets up the stepwise induction schema: induction base case(s), TB, induction hypothesis, TH, and induction conclusion, TC.
    In our example these correspond to the following:

    - *target induction base case* (TB): $\exists u{:}nat\ \exists v{:}nat\ fib(s(0)) = u \wedge fib(0) = v$;
    - *target induction hypothesis* (TH): $\exists u{:}nat\ \exists v{:}nat\ fib(s(x)) = u \wedge fib(x) = v$;
    - *target induction conclusion* (TC): $\exists u' : nat\ \exists v' : nat\ fib(s(s(x))) = u' \wedge fib(s(x)) = v'$.

11. INDUCTION MAPPINGS: Involves witnessing the existential quantifiers at the induction cases of the target schema by matching and then mapping across (sub) structures from the source proof (these may undergo considerable transformation before being used as witnesses):

    - BASE CASE MAPPINGS: Access SB. Map across one on one the witnesses for the source induction base cases and apply the same witnesses at the target

induction base cases so as to witness a (base-case) value for each of the Φ components.

In our example the mapped witnesses for the 2 tuple components, $\exists u' fib(s(0)) = u'$ and $\exists v' fib(0) = v'$, at the target base case are $s(0)$ and $s(0)$.

Simple refinements deal automatically with any *type discrepancy* between source and target base cases *and* with collapsing *multiple* base-cases, in the case of *course_of_values* proofs, into a single stepwise base case.

- STEP CASE MAPPINGS:
  - WITNESSING A VALUE FOR t_1: access SC and substitute the subsidiary calls in TH for those in SC. Result is used as the witness for t_1.
    In our example the substitutions $[s(x)/p(x), x/p(p(x))]$ transform the source induction conclusion into the required target induction conclusion.
  - REMAINING TUPLE COMPONENTS $t_2, ..., t_\Phi$: witnesses for $t_2, ..., t_\Phi$ are provided by target induction hypothesis(ses).
    There is only one remaining tuple component, $\exists u' fib(x) = u'$, at the step case of our example, and a value for u' is witnessed by the target hypothesis $fib(s(x)) = u$.

In effect, to obtain the step case dependencies – dependent subsidiary calls – required to witness a value for the recursive definition, the system accesses, from R_{ind}, the source induction step witness, and reads the dependencies from that.

VERIFICATION COMPONENT OF $G2$

12. Base and step cases are verified by appealing to the source proof lemmas.

Simple transformations may be required of the form:

$$f(x) = g(p(x)) \mapsto f(s(x)) = g(x),$$

where p is the predecessor function and s the successor function. Such transformations convert definitions employing the predecessor to an equivalent formulation that employs the successor function may be required.

SYNTHESIS COMPONENT OF $G1$

13. A value for Y, the output for any X, is provided by the $\exists$ introduction of t_1.
    For the *Fibonacci* output we introduce v as the existential witness (i.e., $fib(x)$).

VERIFICATION COMPONENT OF $G1$

14. That the output Y is indeed t_1 is verified by appealing to $tuple_{def}$, which is a hypothesis at $G1$, and proved through $G2$.

15. Finally, all target sub-goals that require some form of type-checking – or well-formedness check – are entirely satisfied by mapping across the sub-proofs required to satisfy similar sub-goals in the source. These will usually be subgoals of the form 0 *in nat* or *nat in* $u(1)$.

16. TARGET EXTRACTION

    After the automatic application of the completed target rule-tree, R_t, to the identical mapping of the source specification, the OYSTER extraction process extracts a target algorithm from the complete target proof.

5.3.6 Transforming Induction Cases (by Transforming Nested Inductions)

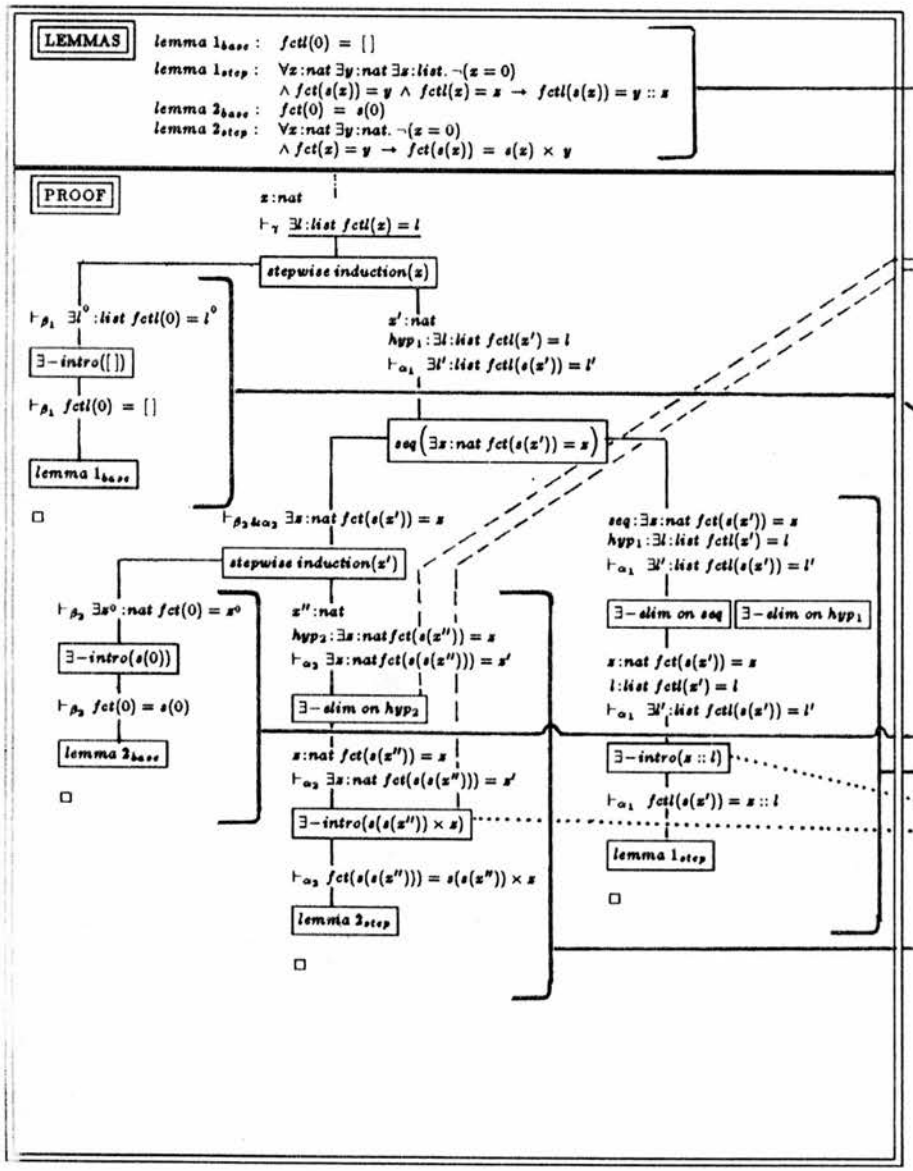
With the *Fibonacci* example, the optimization was achieved through transforming the source induction schema into a different schema with a more efficient computational rule. We now illustrate, by example, how the IPOS is capable of transforming the induction *cases*, and subsequent sub-proofs (§2.3.3), of a source proof in order to optimize an auxiliary recursive program, namely *factlist*. In our example, the transformation is tantamount to transforming a source proof that involves a nested application of induction to a target proof with a single induction. In **fig.** 5–6 we provide a diagram that, as with **fig.** 5–3, depicts the source and target proofs, and the (sub)structure mappings between them.

Finally, we provide an example of a source proof, containing a nested induction, which is optimized by transforming the induction schema associated with the nested induction *and* the induction cases associate with the outermost schema.

The Source to Target Transformation of *factlist*: The Merging of Nested Inductions

Unlike the source synthesis proof for the *Fibonacci* function, the *factlist* function is defined by a stepwise definition – the *factlist* function does not invoke itself more than once at each recursive call – and so is therefore most naturally synthesized

using *stepwise* induction. Recall from §5.2.8, that the recursive step of the factlist definition contains one self recursive call, and a call to the auxiliary *fact* function which is responsible for the redundancy.



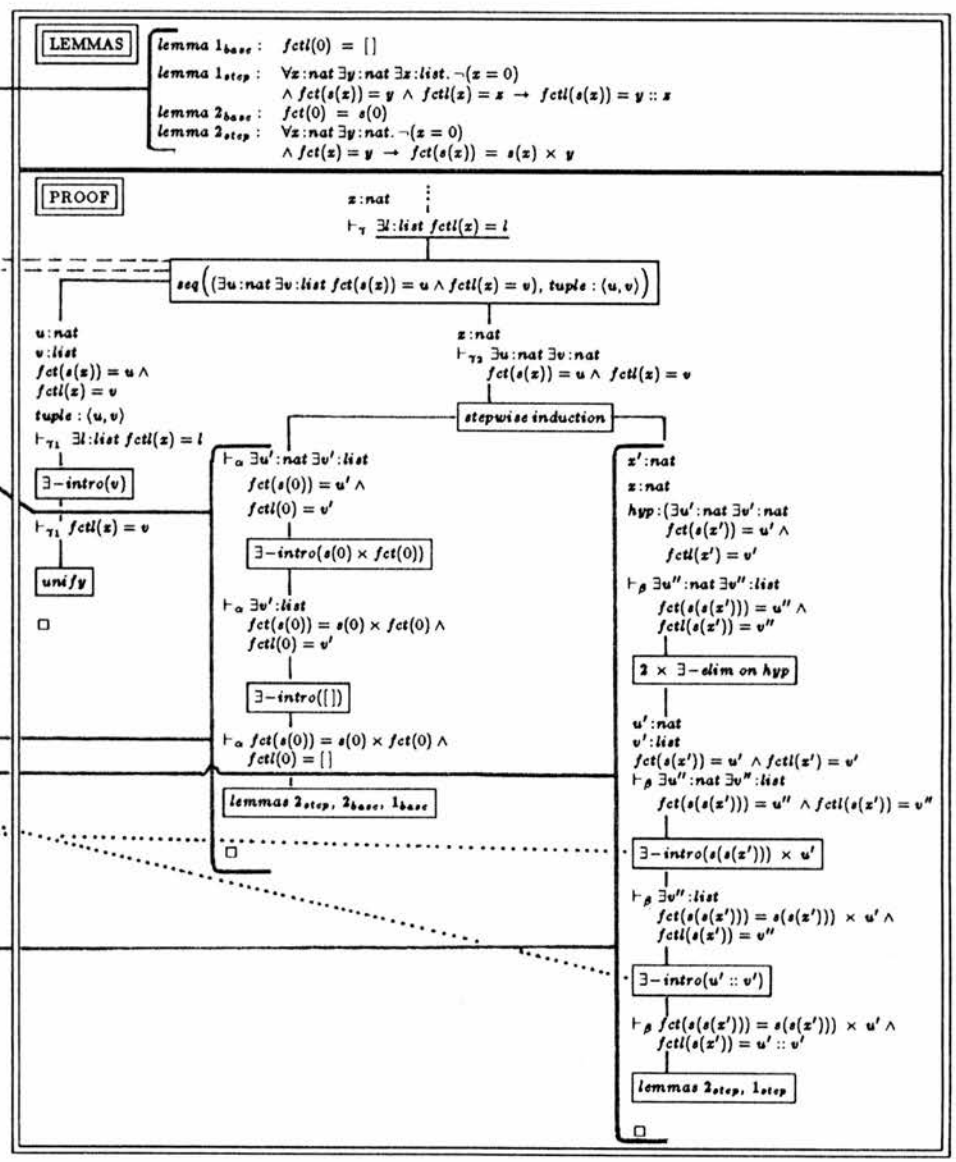


figure 5-6(a): Source Proof for Factlist (employing nested *stepwise* induction)

figure 5-6(b): Target Proof for Factlist (employing single stepwise induction)

Figure 5-6: Mappings between Source and Target *Factlist* Proofs

The *factlist* function is specified thus:

$$\forall{:}x\,\exists l{:}list.fctl(x) = l.$$

The redundancy manifests itself in the source synthesis proof in the form of the nested stepwise induction required to synthesize an extract term for the auxiliary *fact* call (*cf.* **fig.** 5–6(a)). The nested schema means that for each recursive pass corresponding to the outermost induction, the source program must fully recurse on the innermost schema. This is also reflected by the dual nested recursion schema construct of the source proof extract program, a slightly simplified representation of which is shown in **fig.** 5–7 (where the Greek characters correspond to the extract constructions),[15]where *p_ind* signifies the application of *stepwise* induction such that if the induction variable, x' is 0 then the output is *nil*, otherwise the output is $h_2 :: h_1$, where h_1 is the *outermost stepwise* induction hypothesis, and h_2 is the induction hypothesis for the *nested stepwise* induction on x' (regarding **fig.** 5–6, $h_1 = hyp_1$ and $h_2 = hyp_2$). This nested induction is required to establish an output for $fact(x')$, which is then used in the computation, $h_2 :: h_1$ for $factlist(x)$. So the

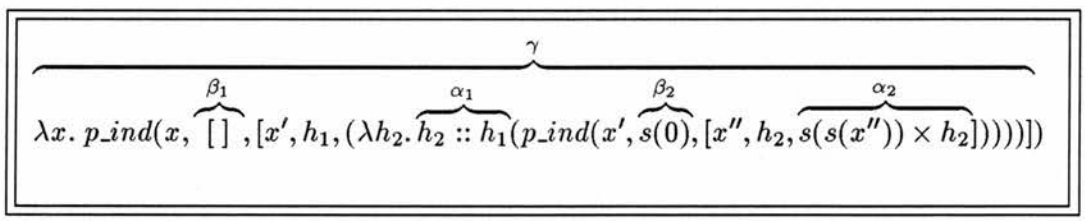


Figure 5–7: The Source Extract for *factlist*

task of the IPOS transformation is to remove this nested induction, and thereby the nested recursion, by effectively specifying the auxiliary call at the level of the outermost induction, and thereby remove the redundancy. This is achieved, as in the case of the *Fibonacci* example, by using proof tupling: having determined the size, Φ, and contents of the required target tuple (see below) the IPOS sequences

[15]Due to the nested inductions, a considerable amount of renaming of variables occurs in the course of the proof (as depicted in **fig.** 5–6). To make the extract program easier to read, we have substituted a single label for variables that denote the same object.

the tuple into the proof. A single stepwise induction is then performed on this tuple, and the existential variables at the induction cases are witnessed by mapping terms across from the instantiated source induction schema.

Such transformations can be characterizaed as transformations on induction *cases* since by removing the innermost (nested) induction we are transforming the cases of the outermost induction.

As with the source to target transformation of self-recursive functions, the optimization of the source auxiliary recursive *factlist* function exploits dependency information contained in the source proof. The step case existential witnesses of the inner and outer inductions of the *factlist* source proof are expressed in terms of the source induction hypotheses (necessarily since the λ-function constructed is recursive). These witnesses are directly exploited in order to satisfy the single step case of the target proof. The mappings between the (sub)structures of the source and target proofs, depicted in **fig.** 5–6, are explained below.[16]

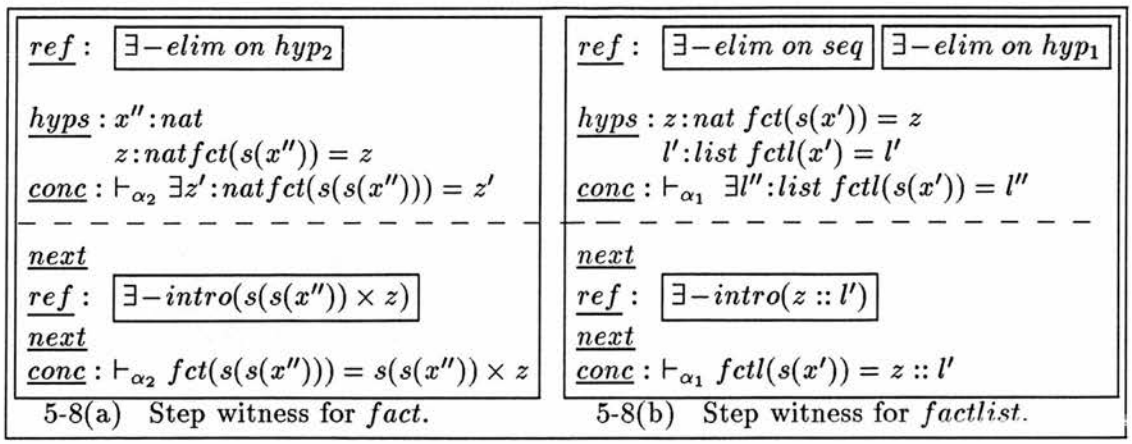
In **fig.** 5–8 below we have represented the witnessing steps of both the source proof inductions. **Fig.** 5–8(a) corresponds to the witnessing of the existential variable at the induction step of the *nested stepwise* schema, and **fig.** 5–8(b) to that of the *outermost stepwise* schema.

The IPOS is able to determine from the above witnessing steps of the source proof, and from the subsequent unfoldings with the lemmas , that the recursive definition of the target tuple requires tabulating two function calls. The first is an occurrence of the auxiliary *fact* function which takes the same argument, n, as in the head of the definition. The other tabulation is a subsidiary *factlist* call which takes the predecessor, $n-1$, of the argument n in the head of the definition. So the recursive definition for the target tuple is

$$factlist_{tup}(s(n)) = \langle fact(s(n)), factlist(n) \rangle.$$

---

[16]For the sake of brevity, we describe the mappings as if they were performed directly on the proofs, were as in practice they are performed on the source proof rule-tree abstraction (i.e., the tactic consisting of the refinements shown in **fig.** 5–6(a)).



5-8(a) Step witness for *fact*.

5-8(b) Step witness for *factlist*.

Figure 5–8: The witnessing steps of the source *factlist* proof

This target definition is given the hypothesis label *tuple* and, as in the *Fibonacci* example, is sequenced into the target proof thus:

$$\big((\exists u{:}nat\, \exists v{:}list\, fct(s(x)) = u \wedge fctl(x) = v),\ tuple : \langle u, v\rangle\big).$$

As in the *Fibonacci* example, the sequencing is performed following the initial ∀-intro application on the main goal (which we ommitted from **fig.** 5–6). Stepwise induction is then performed on the sequenced in goal, where the induction variable is the same as that for the outermost application of induction in source proof.

Hence, since at the stepwise induction step of the target proof we require a tuple definition where the argument n in the head of the definition is equal to $s(s(x))$ then the body of the step case definition is

$$\langle fact(s(s(x))), factlist(s(x))\rangle.$$

Both of these components unfold to terms provided by the proof hypotheses:

- $fact(s(s(x)))$ is equivalent to $s(s(x)) \times fact(s(x))$ where $fact(s(x))$ matches the hypothesis $u = fact(s(x))$. Hence we require a witness value for the first tuple component of $s(s(x)) \times u$. This is obtained by substituting the target hypothesis label u' for z in the step case witness of the *nested* source induction.

- $factlist(s(x))$ is equivalent to $fact(s(x)) :: factlist(x)$ where $fact(s(x))$ matches the hypothesis $u = fact(s(x))$, and where $factlist(x)$ matches the

hypothesis $v = factlist(x)$. Hence we witness a value for the second tuple component of $u :: v$. The IPOS obtains this witness simply by substituting the target hypothesis labels, u' and v', for the labels, z and l', in the step case witness of the *outermost* source induction.

These operations are the means by which the IPOS realizes the tuple satisfaction procedures for auxiliary recursive functions (**A'** and **B'** of §5.2.8).[17]

As with the previous examples, the base case witnesses are mapped across, one on one, from the source, as are the lemma applications required for verifying both the base and step case witnesses.

Extract for Target Factlist

The target program construction is shown in **fig.** 5–9 (where the function *spread* is abbreviated to *spd*).

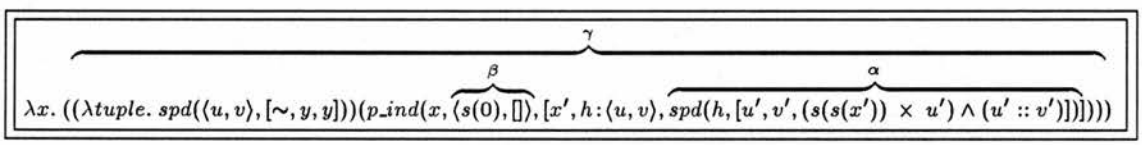


Figure 5–9: The target extract for *factlist*

Note that just as the source proof – **fig.** 5–6(a) – contained two stepwise inductions, with the nested induction being applied at the stepcase of the outermost induction, and the target proof – **fig.** 5–6(b) – contains only a single induction (on a tuple structure), so the source extract program – **fig.** 5–7 – contains a dual nested recursion schema, with the nested recursion being applied at the stepcase

[17]Note that although the tuple size and components can be determined by the IPOS through the tuple satisfaction procedures, this initial analysis is not essential since the heuristic tupling procedure, T2, will suffice (i.e., we need only map across directly the source witnesses at the inner and outer stepwise induction steps in order to form the two tuple components).

of the outermost recursion, and the target extract program – **fig.** 5–9 – contains only a single dual recursion (on a tuple structure).

Transforming Both Induction Schemas and Induction Cases

Recall from §5.2.9, that we introduced a variant of $factlist$ thus,

$$factlist(s(n)) = fact_2(s(n)) :: factlist(n),$$

were the auxiliary function $fact_2$ is a $2-step$ recursive function thus:

$$\begin{aligned} fact(0) &= 1; \\ fact(s(0)) &= 1; \\ fact(s(s(n))) &= s(s(n)) \times fact(n), \end{aligned}$$

and where the IPOS would construct a target tuple of length 3, where one component is the subsidiary $factlist$ call and we would need to tabulate 2 subsidiary calls for the $fact$ computation.

$2-step$ induction is best suited to construct the auxiliary $fact_2$ function since $fact_2$ is naturally a $2-step$ definition. The schema for $2-step$ induction is as follows:

$$\frac{\vdash P(0) \qquad \vdash P(s(0)) \qquad \forall v : pnat.\ P(v) \vdash P(s(s(v)))}{\vdash \forall x : pnat.\ P(x)},$$

and the source proof node at the induction step case, resulting from the (nested) application of $2-step$ induction on the (sequenced in) $fact_2$ goal, would be almost identical to **fig.** 5–8(a). The only difference is that the recursive argument in the goal conclusion is two, rather than one, applications of the successor function out of step with the recursion argument in the induction hypothesis. We show this node in **fig.** 5–10 below (where we have also indicted the application of the (nested) $2-step$ induction prior to elimination on the $2-step$ induction hypothesis hyp_{2-step}).

To perform the proof tupling transformations on such a nested induction, the IPOS would need to tabulate 2 $fact_2$ function calls, along with the $factlist$ call. That the target tuple includes 2 $fact_2$ function calls is determined by precisely

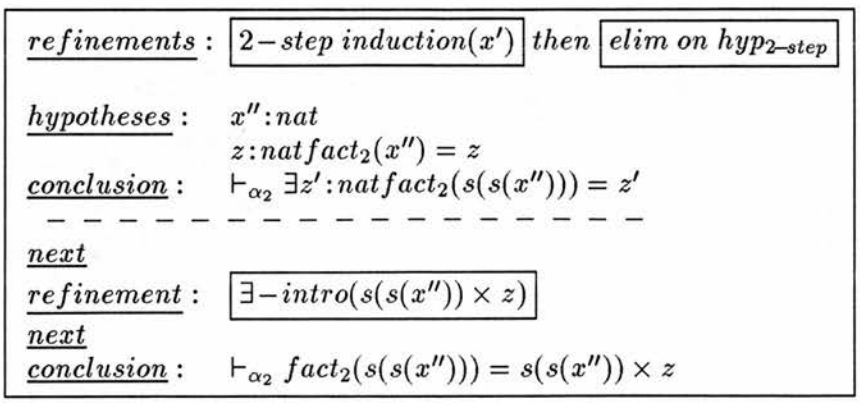


Figure 5–10: Witnessing step of $fact_2$ (nested proof).

the same reasoning that is used to form a target tuple for the *Fibonacci* example: the body of the step case definition for $fact_2$ contains a self recursive call to $fact_2$ that is 2 applications of the common generator function, in this case s, out of step with the head of the definition. This is clearly illustrated by replacing z in the *next conlusion* slot, of **fig.** 5–10, by the hypothesis that it labels thus:

$$\vdash_{\alpha_2} fact_2(s(s(x'')))) = s(s(x'')) \times fact_2(x).$$

Hence, to optimize the $fact_2$ function we would require a tuple of two components (i.e., $\Phi = 2$), where the tabulations would correspond to $fact_2(s(n))$ and $fact_2(n)$. Hence, if $fact_2$ appears as the auxiliary function call of $factlist$, then the required target tuple would contain three components (i.e., $\Phi = 3$), and we would sequence the following goal into the target proof:

$$\big((\exists u{:}nat\ \exists v{:}nat\ \exists w{:}list\ fact_2(s(x)) = u \wedge fact_2(x) = v \wedge fctl(x) = w),\ tuple : \langle u, v, w\rangle\big).$$

Note that, in effect, in performing the above source to target transformation we have *both*:

- transformed a source proof with a nested induction to a target proof with a single induction (employed on a tuple); and
- in doing so, transformed the (nested) $2-step$ induction into a standard ($1-step$) stepwise induction.

Hence proof tupling on source proofs that contain a nested induction structure, where either of the inductions is in itself susceptible to optimization through tupling, is tantamount to combining the transformation of induction schemas with the transformation of induction cases.

5.4 Merits of Proof Tupling and Comparisons with Program Tupling Transformations

In *Chapter 3* we reviewed one of the most influential strategies for program transformation, the *fold/unfold* technique. This was primarily discussed within the context of Darlington's NLP program transformation system, and Chin's use of the *fold/unfold* strategy to perform tupling transformations.

We identified two key steps for transformation using the *fold/unfold* strategy. These steps correspond to the most difficult aspects as far as *automation* is concerned, and in NLP, and similar systems, require some form of user guidance:

- Lemma generation: the introduction of an appropriate function definition in terms of the source definition (the so-called *Eureka step*).

- Folding: When to fold the tuple function definition with the source definition.

Within the context of NLP type tupling transformations the attainment of these key steps involves the following:

1. Tuple creation: the introduction of an appropriate tuple function definition in terms of the source definition (the so-called *Eureka step*). This entails employing special purpose procedures for analysing *symbolic dependency graphs* so as to:

   (a) create a tuple structure of the correct size (i.e., determining the number of subsidiary calls to be tabulated); and

(b) determine what the contents of the tuple are (i.e., which subsidiary calls in the source definition are tabulated within the target tuple).

2. Folding: matching is used as a means of testing for the successful folding of the tuple function definition with the source definition.

3. Target verification: separate correctness proofs are required to verify that the target of any tupling transformation is correct with respect to its, and the source's, specification.

We now discuss the differences, and advantages, that the IPOS approach to optimization has on the exploitation of dependency information, correctness, and control and search issues.

5.4.1 The Reduced Workload Regarding Dependency Analyses

The rule-tree abstractions are designed to preserve precisely that information which proofs contain in addition to the programs that they synthesize:

- a description of the task being performed;
- a verification of the method; and
- an account of the dependencies between facts involved in the computation.

The fact that they contain an account of the dependencies between the facts involved in the computation means that, in order to construct an appropriate recursive definition for the target, we do not have to appeal to complex tuple formation procedures that construct *symbolic dependency graphs*, DG's, and subsequently analyse these in order to form the requisite *Eureka* tuple.

This is clearly the case for any explicit definition of the target tuple constructed through the heuristic procedure T2: if T2 is applicable then a direct one to one

mapping of the source recursive definition will provide the requisite explicit definition for the target tuple, without *any* need to consult or abstract dependency information.

Regarding T1, and the construction of a recursive definition of the target tuple at the step case of target stepwise induction, let us first briefly summarize the essentials of the dependency analyses involved in the program tupling systems. Recall from *Chapter 3* that with the optimization of the *Fibonacci* function, Darlington, and later Chin, use ideas taken from (Pettorossi, 1984) in order to find a pair of *matching tuples* by the unfolding of selected calls to the source program, and then using matching as a means of testing for successful folding. The first of these matching tuples is then selected for the target (*Eureka*) tuple. This process actually involves constructing a representation of the function's evaluation tree which shows the calling structure of the subsidiary recursive calls, and then analysing the recursive calls to find the matching tuples.

Returning to the *Fibonacci* example and starting with the main function call, Chin's analysis replaces $fib(n)$, the first cut, with its two subsidiary calls, $\langle fib(n-1), fib(n-2)\rangle$. This gives us the second cut. The analysis then proceeds by unfolding only that call in a cut which is *not* a subsidiary call of the other call, i.e., the topmost item. So, since the function call $fib(n-2)$ is a subsidiary call of $fib(n-1)$, then only $fib(n-1)$ is unfolded. This provides the third cut, $\langle fib(n-2), fib(n-3)\rangle$, of the DG. The third cut matches the second cut, thus providing the analysis with a matching tuple.[18]

Such an analysis tells us two things:

1. firstly, *the number*, Φ, of subsidiary calls of the main function calls required to form the tuple (i.e., the determination of the tuple size); and

2. secondly, *which* subsidiary calls are to be tabulated.

---

[18]Recall from *Chapter 3*, that a pair of cuts are said to *match* if a consistent substitution can be obtained when each function call of the first cut is matched with the corresponding function call of the second cut.

An advantage of *proof tupling* is that *both* of these things, required for the tuple formation, are contained in the source proof, and preserved in the source proof rule tree abstraction. This means that they can readily be abstracted from the proof and exploited for the construction of the target tuple *without* any additional dependency graph construction and analysis procedures. We described this process, throughout §*5.3*, using *Fibonacci* as our standard example.

An interesting point to note is that Chin's analysis of unfolding cuts, followed by matching (or, indeed, unification) is very similar to the process of rippling out followed by fertilization during synthesis. The analogy nicely illustrates the fact that the information required by Chin's tuple analysis is (explicitly) present within our synthesis proofs.

5.4.2 Correctness

Recall from *Chapter 3* that the original fold/unfold strategy, as it was presented in (Burstall & Darlington, 1977b) was not provided with a *correctness guarantee* for the source to target transformations. Later incarnations have been shown to be have a correctness guarantee for specified classes of functions (notably (Tamaki & Sato, 1984) and (Chin, 1990)). However, each extension to the class of functions requires a corresponding extension to the correctness procedures, and this leads to a considerable work overhead (proportional to the range of transformations – or *generality* – of the system).

This is not a problem regarding the IPOS, and any future extensions thereof. We can summarize what has been said regarding the correctness of the IPOS transformations as follows:

- Extract programs are correct with respect to the complete specifications of the synthesis proofs from which they are extracted.
- Given the respective specifications, the target and source rule-trees contain all the information required to construct the respective proofs.
- The respective specifications are, in the case of optimization, the same.

- This is achieved by ensuring that the proof refinement (sub)tree associated with the alternative (target) means of computing the input/output relation specified in the source specification is *sequenced* into the target proof as a new sub-goal. Thus the main goal of the target proof remains identical to the source specification.

- Therefore, correctness of the target proof *and* of the source to target transformation is ensured.

Hence the correctness of *all* terminating transformations is ensured *without* having to additionally provide, or extend, any correctness criteria, or proof, each time we extend the range of programs to which the transformations are applicable.

5.4.3 Search

The fact that the IPOS transformation tactics are (partially) specified at the meta-level, in terms of syntactic pre- and post-conditions, reduces the amount of search that would be involved if the target proof where constructed at the object-level. In other words, since we can regard the rule-trees, together with pre- and post-conditions, as proof plans then a general advantage of performing *tactic transformations* – i.e., meta-level transformations on the object-level tactics – is that the transformation space is equivalent to a planning search space which is far smaller than the object-level search space.

Further factors which play a beneficial role regarding search and control include:

- the way that dependencies are sought during tupling transformations;

- the means by which the target recursive step is completed;

- and the form of equation development used all have a significant effect on the amount of search involved during the transformation.

We shall consider in turn how the IPOS reduces the search involved with each of these three factors in comparison with previous program tupling systems (Pettorossi, 1984; Burstall & Darlington 1977a; Chin, 1990).

Searching For Dependencies

Note that no extensive search is involved in the analysis of the source proof in order to determine Φ and to witness a value for the tuple components. The portions of the source proof that are accessed for the analysis correspond to specific semantic units: the specification, the application of induction, the induction base and step cases, the unfolding step, and the witnessing rule. These are clearly represented as distinct sub-lists within the rule-tree abstractions, and the IPOS knows precisely where to look in order to access any of the aforementioned units. For example, the induction step will always correspond to that rule applied at the deepest node of the decision tree employed to separate the various cases (*cf.* **fig.** 5–10). So, within the rule-tree, the induction step occurs as the last case of a nested case analysis.

So, unlike program tupling, the IPOS proof tupling optimizations do not require the construction of a (potentially infinite) dependency graph, nor does it require any procedures for searching the dependency graph in order to find a *matching tuple.*

Searching for a Fold

The motivation that drives the tuple formation procedures also has a considerable bearing on the amount of search involved during tupling transformations: Darlington's NLP, and Chin's HOPE$^+$, tuple analysis is motivated by the desire to find a tuple which can be used for *folding.* This involves quite extensive search in order to find a successful fold. For example, during the optimization of the *Fibonacci* procedure, the NLP system performs numerous unfoldings in order to obtain the following explicit definition for the target *Eureka* tuple:

$$g(n+1) \;=\; \langle fib(n+1) + fib(n), fib(n+1) \rangle.$$

The system *must* then attempt to find a fold so as to introduce a recursion into the target equations. In the NLP system, the search for a fold requires considerable user interaction, and involves observing that all the components necessary to match the above equation are present within the initial definition for the auxiliary function g:

$$g(n) \;=\; \langle fib(n+1), fib(n)\rangle.$$

Hence, the explicit definition is re-written using *where abstraction*, *cf. Chapter 3*, to the following:

$$g(n+1) \;=\; \langle u1+u2, u1\rangle \;\; where \;\; \langle u1, u2\rangle \;=\; \langle fib(n+1), fib(n)\rangle,$$

which easily folds with the initial definition yielding the desired optimized function *recursive* definition:

$$g(n+1) \;=\; \langle u1+u2, u1\rangle \;\; where \;\; \langle u1, u2\rangle \;=\; g(n).$$

The IPOS analysis, on the other hand, is motivated by the desire to find witnesses for the tuple components at the induction step of a synthesis *proof*. Once this has been achieved, by the aforementioned ways, then the proof is completed in much the same way as any inductive synthesis proof: by a process of *unfolding* until all terms in the conclusion match terms in the proof hypotheses. This process, as discussed in *Chapter 4*, is readily susceptible to automation.

The Form of the Target Equation Development

A graphic way of illustrating both

- why folding is an essential ingredient of NLP style equation transformations, and
- how the IPOS approach to tupling transformations *reduces* the associated search space,

is obtained by comparing the //-form of equation developments associated with the IPOS transformations with the more conventional NLP style equation development.

Within NLP type systems the head of the developing equations remains constant, and it is only the body that is modified, i.e., re-write rules are only applied to the *left hand side* of equations. This form of equation development we shall refer to as ⊔-form.[19] We shall again use fib_{tup} to represent the tuple function responsible for collapsing the two recursive calls in the standard *Fibonacci* definition into a single tuple function.

$$
\begin{array}{lcll}
fib_{tup}(n) & = & \langle fib(s(n)), fib(n)\rangle & \\
fib_{tup}(0) & = & \langle 1,1\rangle & \textbf{unfold } fib \\
 & & & \\
fib_{tup}(s(n)) & = & \langle fib(s(n)) + fib(n), fib(s(n))\rangle & \textbf{unfold } fib \\
 & = & \langle u+v, u\rangle \; where \; \langle u,v\rangle \; = \; \langle fib(s(n)), fib(n)\rangle & \textbf{abstract} \\
 & = & \langle u+v, u\rangle \; where \; \langle u,v\rangle \; = \; fib_{tup}(n) & \underline{\textbf{fold}}\ fib_{tup}
\end{array}
$$

This *form* of equation development, together with the formal definition of folding (stated in *Chapter 3* and repeated below):

> If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F''; then add the equation $F = F''$,

means that since throughout the equation development we always retain the same equation head that therefore *folding* with the source equations *must* be introduced at some point in order to introduce a recursion into the tail of the developing equations. There is not, however, any procedure for knowing *when* to fold (or when to *forced fold*: the combination of *where* abstraction and folding). NLP does not have the bi-directional development process of both sides of the developing equations. This presents control problems, and is one primary reason why user guidance is usually required in such systems in order to avoid flawed attempts at

[19]The term ⊔-form was originally coined by Alan Bundy in an informal departmental note which investigated the re-formation of the well established fold/unfold technique in terms of repeated unfoldings.

folding (the other reason being the *eureka* step corresponding to the generation of the auxiliary tuple).

The main advantages of the //-form process exhibited by the above, and subsequent, examples are that: the process develops in a bi-directional manner in that both sides of the recursive step equation can be re-written in the search for matching (unifiable) terms.

This "parallel" development of both head and body towards a "unifiable" pattern, such that induction terms may be eliminated, is a useful methodology since the modifications on both sides of the developing equation can, by a process of feedback, direct each other. Furthermore, since the //-form development is directed towards unifying terms on either side of the equation, and since we can modify both sides of the equation we can avoid the decision(s) as to when, and with what, to fold. We can limit the //-form to unfolding. So, in addition to the source proof providing information to guide the unfolding, the //-form development also reduces the search space.

5.5 Further Work: Linear to Logarithmic Complexity Proof Transformations (an Extension to the IPOS)

Leaving the specialization application aside, the IPOS is currently limited in the scope of inductive proofs that it can transform, and thereby limited in the scope of recursive programs that it can optimize. This is because the present implementation is geared toward proof tupling transformations and, recalling §5.2.2, the following precondition for proof tupling is most naturally realized within a *course_of_values* proof:

> There exist two or more induction terms, $f(n), ..., f(n-i)$, which share some *common induction variable(s)* in a function definition (where $i \geq 2$).

It was the author's original intention to augment the IPOS with a further class of source to target transformations on inductive proofs: *stepwise* induction to *divide_and_conquer* induction. This would enable the automatic transformation of linear procedures, such as that generated by the *stepwise Fibonacci* extract, into logarithmic procedures. So if such an extension were implemented the IPOS would be capable of automatically transforming exponential procedures to logarithmic procedures through the proof transformation of linear procedures. Using *Fibonacci* as our example this can be depicted by the table in **fig.** 5–11.

Such an extension was not possible due to time constraints, although all the essential machinery such as the formation of rule-trees and the various accessing, mapping and transformation operators, would not require any extensive alteration to achieve the task.

PROOF PROPERTIES	SOURCE 1	TARGET 1/SOURCE 2	TARGET 2
INDUCTION SCHEMA	COURSE-OF-VALUES	STEPWISE	DIVIDE & CONQUER
TYPE OF IND. VARIABLE	NATURAL NUMBER	TUPLE	MATRIX
PROCEDURE COMPLEXITY	EXPONENTIAL	LINEAR	LOGARITHMIC
MEANS OF CONSTRUCTION	MANUAL SYNTH.	PROOF TUPLING	MATRIX MULT.

Figure 5–11: Relation between source and target proofs of (extended) IPOS.

We shall first outline how the *divide_and_conquer* induction rule can be used to construct a logarithmic procedure for computing the *Fibonacci* numbers. Secondly, we shall sketch the fundamentals of attaining such a procedure through transforming the *stepwise* proof (we shall here, as we did throughout §5.2, concentrate on a *general* transformation methodology, of which the optimization of the linear *Fibonacci* procedure would be an instance). We then provide a step by step procedure for transforming a *course_of_values* procedure through to a logarithmic procedure (with the linear procedure occurring as an intermediary state). Finally, we provide a simple example of a linear to logarithmic program transformation through the extended IPOS proof transformations.

The basic idea is as follows, we would use the method of *matrix multiplication*, rather than tupling, to transform the *stepwise* proof into a *divide_and_conquer* proof that yields a procedure for computing *Fibonacci* with *logarithmic* complexity.

5.5.1 Using the *Divide_and_Conquer* Schema to *Synthesize* Logarithmic Recursion

A *logarithmic* procedure for calculating the *Fibonacci* numbers is synthesized using the (constructor) *divide_and_conquer* induction schema:

$$\frac{\vdash \forall x : nat.\ P(x)}{\vdash P(0) \qquad \forall v : nat.\ P(v) \vdash P(v+v) \qquad \forall v : nat.P(v) \vdash P(s(v+v))}.$$

The key points of the synthesis are as follows. A logarithmic procedure for computing the numbers *Fibonacci* is most naturally synthesized by expressing the tuple $fib_{tup}(n)$, for calculating *Fibonacci* through a linear procedure, in terms of matrices:

$$fib_{tup}(n) \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n fib_{tup}(0)\ .$$

So we can calculate $fib_{tup}(n)$ by calculating $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n fib_{tup}(0)$.

That is, we can calculate the *Fibonacci* numbers by appealing to the n^{th} power of a matrix, and this is naturally a *logarithmic* procedure.

Letting $m(n) \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ then by using *matrix multiplication* the base and two step cases of the *divide_and conquer* induction schema are witnessed by the following values for $fib_{tup}(0)$, $fib_{tup}(2n)$ and $fib_{tup}(2n+1)$:

(3) $fib_{tup}(n) \;=\; m(n) \times fib_{tup}(0)$;

(4) $fib_{tup}(2n) \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} fib_{tup}(0) \;=\; (m(n))^2 \times fib_{tup}(0)$; and

(5) $fib_{tup}(2n+1) \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n+1} fib_{tup}(0) \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} (m(n))^2 \times fib_{tup}(0).$

Letting "*there exists an object, m, of type matrix*" be denoted by $\exists m : mat$, we instantiate the induction schema as illustrated in **fig.** 5–12 (for the purpose

$$\vdash \forall x : nat.\ \exists m{:}mat.\ m(x) \times fib_{tup}(0) = fib_{tup}(x)$$

$$\vdash \exists m{:}mat.\ m \times fib_{tup}(0) = fib_{tup}(0)$$

$$\exists m{:}mat.\ m \times fib_{tup}(0) = fib_{tup}(v) \ \vdash\ \exists m'{:}mat.\ m' \times fib_{tup}(0) = fib_{tup}(v+v)$$

$$\exists m{:}mat.\ m \times fib_{tup}(0) = fib_{tup}(v) \ \vdash\ \exists m'{:}mat.\ m' \times fib_{tup}(0) = fib_{tup}(s(v+v))$$

Figure 5–12: The instantiated *devide_and_conquer* schema

of presentation we have stacked the base and two step cases of the schema rather than present them on a single line). The base case follows from the evaluation of $fib_{tup}(0)$: since $fib_{tup}(0) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, then

$$m \ = \ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The first step case is satisfied by witnessing the value for m' provided by (**3**) and (**4**). That is,

$$m' \ = \ m^2.$$

The second step case is satisfied by witnessing the value for m' provided by (**3**) and (**5**). That is,

$$m' \ = \ \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times m^2.$$

In each case we can verify the instantiations for m' in the same manner in which we derived (4) and (5) *given the initial* linear *version* (**1**).

Remark

An important point to note is that we can *only* reduce the problem of calculating the *Fibonacci* numbers by evaluating the n^{th} power of a matrix *if* we have available the initial *linear* recursive definition for *Fibonacci* in terms of tuples. Hence, if our

starting point is the standard *course_of_values* definition then we *first* need to convert this into the linear version in order to obtain the *logarithmic* version.

5.5.2 Linear to Logarithmic Transformation

In order to *automate* the process *for any* given bi-linear function, f, which, like the *course_of_values Fibonacci* definition, fits the schematic function for common generator functions ($EQ.1$, §5.2.2) *and* can be linearized through the GPPT (§5.2.6) we require a general rule for creating the correct matrix, $\mathcal{M}$, for f. In general $f_{tup}(n) = \mathcal{M}(n) \times f_{tup}(0)$, where f_{tup} is the tuple function which linearizes f, and $\mathcal{M}(n) \equiv \mathcal{M}^n$. The task for the automatic transformation of the linear version to the logarithmic version is to calculate $\mathcal{M}$ automatically. Again, the tuple size, Φ, is seen to be the crucial factor. Once $\mathcal{M}$ has been determined then the induction schema is satisfied in precisely the same way as it was for the *Fibonacci* function, where

$$\begin{array}{lrcl} \underline{\text{induction step 1:}} & f_{tup}(2n) & = & (\mathcal{M}(n))^2 \times f_{tup}(0), \\ \underline{\text{induction step 2:}} & f_{tup}(2n+1) & = & (\mathcal{M} \times (\mathcal{M}(n))^2) \times f_{tup}(0). \end{array}$$

Clearly, once this task has been automated the source to target transformation of an exponential procedure to a logarithmic one, for some function f, will also be performed automatically, with the linear procedure being produced, and extracted if desired, as an intermediary, *and* necessary, state of the proof transformation.

Using the formulation of the *general procedure for tupling*, the explicit definition for the linear procedure was given, in step 3, as

$$\textbf{(1)}: \; f_{tup}(n) \;=\; \langle f(\delta^1(n)), f(\delta^2(n)), f(\delta^3(n)), ..., f(\delta^\Phi(n)) \rangle.$$

where Φ is the tuple size (if the initial, source, definition was *not* the *linear* version but the *exponential* version, then Φ must be initially determined by the *general procedure* steps 1 and 2). Note that **(1)** is, in fact, the linear procedure obtained, through the proof tupling, from a function definition that fits the general schema $EQ.1$. We shall, for presentations sake, remain with **(1)** to describe the transformation of linear to logarithmic procedures. Following this description, we provide the simple extension required, in the formation of the requisite matrix, to trans-

form linear algorithms that where obtained from function definitions that fit the more general schema $EQ.1'$ (§5.2.6).

Letting n be any of the naturals, the general schematic tuple function, **(1)**, for calculating f through a linear procedure can be expressed in terms of matrices as shown below, where the subscripts denote the rows and columns of the matrix. The form of the matrix reflects how the identity of each component within the tuple $f_{tup}(n-1)$ is related to each component within the tuple $f_{tup}(n)$:

$$\textbf{(2)}\quad f_{tup}(n) = \begin{pmatrix} f(\delta^1(n)) \\ f(\delta^2(n)) \\ f(\delta^3(n)) \\ \cdots \\ f(\delta^{\Phi-1}(n)) \\ f(\delta^{\Phi}(n)) \end{pmatrix} = \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_{\Phi} \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_{\Phi} & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} f(\delta^2(n)) \\ f(\delta^3(n)) \\ f(\delta^4(n)) \\ \cdots \\ f(\delta^{\Phi}(n)) \\ f(\delta^{\Phi+1}(n)) \end{pmatrix}$$

$$= \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_{\Phi} \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_{\Phi} & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix} f_{tup}(n-1).$$

Hence the corresponding matrix for the general schematic tuple function, f_{tup}, is a $\Phi \times \Phi$ matrix of the form shown below:

$$\textbf{(3)}\quad f_{tup}(n) = \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_{\Phi} \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_{\Phi} & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix}^n \times f_{tup}(0).$$

Since we have let

$$\textbf{(4)}\quad \mathcal{M}(n) = \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_{\Phi} \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_{\Phi} & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix}^n,$$

then the following holds,

$$(5)\quad f_{tup}(n) = \mathcal{M}(n) \times f_{tup}(0),$$

$$(6)\quad f_{tup}(2n) = \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_\Phi \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_\Phi & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix}^{2n} f_{tup}(0)$$

$$= (\mathcal{M}(n))^2 \times f_{tup}(0),$$

$$(7)\quad f_{tup}(2n+1) = \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_\Phi \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_\Phi & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix}^{2n+1}$$

$$= \begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_\Phi \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_\Phi & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix} (\mathcal{M}(n))^2 \times f_{tup}(0).$$

So once Φ is known, either by being given the linear version of f as the source *or* deriving the linear version from the exponential version through the proof tupling procedures, then the derivation of a logarithmic procedure is purely mechanical: the extended IPOS would automatically sequence into a proof, of the conjecture specifiying $f(n)$, the corresponding matrix object. *Divide_and_conquer* induction is then applied, in place of the *stepwise* induction, to the *same* induction variable, n, as in the linear proof. **(3)**, or equivalently **5**, correspond to the induction hypothesis,

$$\exists \mathcal{M}' : mat.\ \mathcal{M}' \times f_{tup}(0) = f_{tup}(n),$$

after existential elimination on $\mathcal{M}(n)$. The induction cases are satisfied as follows:

- The **base case**,

$$\vdash \exists \mathcal{M} : mat.\ \mathcal{M} \times f_{tup}(0) = f_{tup}(0),$$

follows directly from the evaluation of $f_{tup}(0)$. i.e., the base case is instantiated as follows:

$$\begin{bmatrix} 1 & 0 & 0 & . & 0 & 0 \\ 0 & 1 & 0 & . & 0 & 0 \\ . & . & . & . & . & . \\ 0 & 0 & 0 & . & 1 & 0 \\ 0 & 0 & 0 & . & 0 & 1 \end{bmatrix} \times f_{tup}(0) = f_{tup}(0).$$

- The **first step case** is as follows,

$$\exists \mathcal{M}' : mat.\, \mathcal{M}' \times f_{tup}(0) = f_{tup}(n) \vdash \exists \mathcal{M}'' : mat.\, \mathcal{M}'' \times f_{tup}(0) = f_{tup}(2n).$$

After performing existential elimination on the hypothesis part of the step case we obtain:

$$\mathcal{M}(n) \times f_{tup}(0) = f_{tup}(n) \vdash \exists \mathcal{M}'' : mat.\, \mathcal{M}'' \times f_{tup}(0) = f_{tup}(2n).$$

We then witness a value for the existential quantified variable, $\mathcal{M}''$, in the conclusion part of the step case with $\mathcal{M}^2$. I.e.,

$$\textbf{(8)} \quad (\mathcal{M}(n))^2 \times f_{tup}(0) = f_{tup}(2n).$$

This is simply equation **(6)** re-written in a right to left direction. So **(8)** is automatically verified by appealing, respectively and in a right to left direction, to equations **(6)** and **(5)** so as to unify the conclusion, **(8)**, with the hypothesis, **(3)**.

- Similarly, the **second step case**

$$\exists \mathcal{M}' : mat.$$
$$\mathcal{M}'(n) \times f_{tup}(0) = f_{tup}(n) \vdash \exists\, \mathcal{M}'' : mat. \mathcal{M}''(n) \times f_{tup}(0) = f_{tup}(s(2n))$$

is satisfied by performing existential elimination on the hypothesis part of th e second step case, and then witnessing a value for the existential quantified variable, $\mathcal{M}''$, with $\mathcal{M} \times \mathcal{M}(n))^2$ to obtain the following sub-goal:

$$(9)\quad \left(\begin{bmatrix} 1_1 & 0_2 & 0_3 & . & . & 0_{\Phi-2} & 0_{\Phi-1} & 1_\Phi \\ 1_2 & 0 & 0 & . & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0_{\Phi-1} & 0 & 0 & . & . & 1 & 0 & 0 \\ 0_\Phi & 0 & 0 & . & . & 0 & 1 & 0 \end{bmatrix} \times (\mathcal{M}(n))^2 \right) \times f_{tup}(0) = f_{tup}(s(2n)).$$

Similarly, this is equation **(7)** re-written in a right to left direction. So **(9)** is automatically verified by appealing, respectively and in a right to left direction, to equations **(7)** and **(5)** so as to unify the conclusion, **(9)** with the hypothesis **(3)**.

The Matrix (Sub)Proof for Logarithmically Computing *Fibonacci*

In **fig.** 5–13 we display the sequenced-in target (sub)proof for constructing the matrix function using *divide_and_conquer* induction. In practice, the target (sub)proof, **fig.** 5–12, of the linear (*stepwise*) to logarithmic (*divide_and_conquer*) transformation would appear nested within the sequenced goal of our target proof, **fig.** 5–3(b), for the exponential (*course_of_values*) to linear (*stepwise*) transformation, i.e.,[20]

$$A.\ \ seq\big((\exists u{:}nat\, \exists v{:}nat\, fib(s(x)) = u \wedge fib(x) = v),\ tuple : \langle u, v\rangle\big).$$

The main goal of the (sub)proof is that of §5.5.1:

$$B.\ \ \vdash_\gamma\ \forall x : nat\, \exists m' : mat.\, m'(x) \times g(0) = g(x),$$

where $g = \langle u, v\rangle$, i.e., the tuple object sequenced in through A above.

Pre- and Post-Conditions for Linear to Logarithmic Transformation

Just as the pre-conditions for the proof tupling transformations included the presence of two or more induction terms which share some common induction variable(s), so the pre-conditions for the matrix multiplication transformations would include the following:

---

[20]So the complete logarithmic target proof will have a nested sequencing structure two levels deep, where the first sequenced goal is A, and the second sequenced goal B.

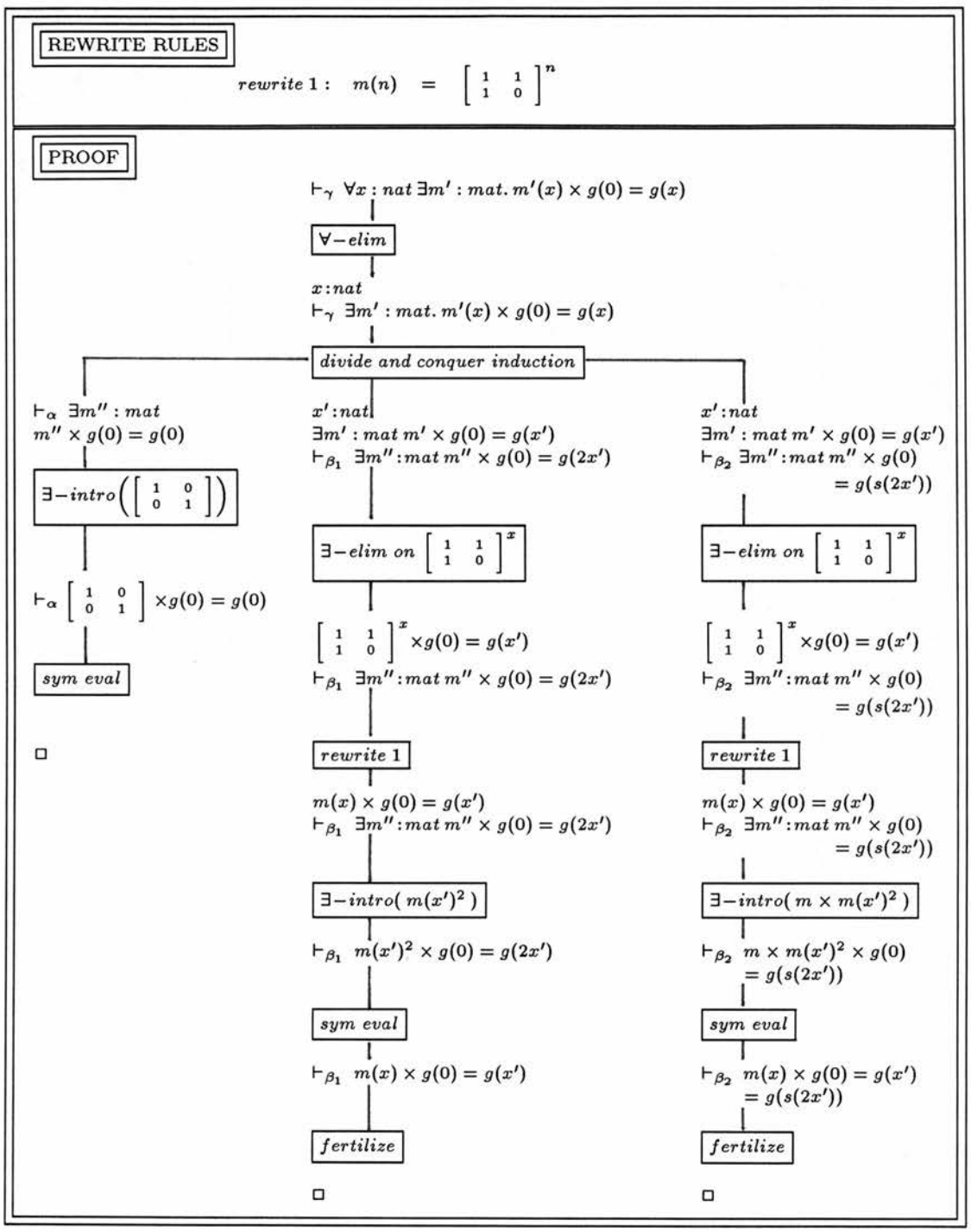


Figure 5–13: Sequenced target (sub)proof for *Fibonacci* using *divide_and_conquer* induction on matrices.

the presence of a fixed sized tuple within which common subsidiary function calls arising from the unfoldings of each of $f'(n), ..., f'(n-i)$ are merged (thus forming a recursive function without the *course_of_values* redundancy).

That is the precondition for the matrix multiplication transformations – or linear to logarithmic transformations – would be the post-conditions of the proof tupling transformations (§5.2.2). This illustrates how, by "dove-tailing" each of the source to target transformations, depicted in **fig.** 5–11, the passage from an exponential procedure to a logarithmic one can be achieved completely automatically (and with the correctness guarantee afforded by the specification language) through proof transformation.

We do wish, however, to emphasize that, although the linear to logarithmic transformation follows naturally from the exponential to linear transformation, the two are distinct. That is, the preconditions of the matrix multiplication transformations would not include a prior proof tupling transformation, although the post-conditions of the latter will match with the pre-conditions of the former.

Extended General Procedure for Exponential to Logarithmic Transformation

The automatic transformation of a specific exponential procedure for evaluating $f'(n)$, through to the logarithmic procedure can be summarized by the following steps:

1. Input the course_of_values proof.

2. Follow the *General //-form Tupling Procedure* to obtain the stepwise proof, incorporating a tuple structure of size Φ such that $f'_{tup} = \langle f'(n-1), f'(n-2), ..., f'(n-\Phi)\rangle$.

3. Extract the linear program if desired.

4. Construct a specific instance, m, of the $\Phi \times \Phi$ schematic matrix shown in **2.**

5. Introduce m into the stepwise proof by *sequencing* the following

$$H: \quad f'_{tup}(n) \;=\; m(n) \times f'_{tup}(0),$$

before the stepwise induction application to induction variable n. This automatically removes all the proof-tree below the sequenced node *and* ensures that the logarithmic target will have the same functionality – satisfy the same specification – as the linear (and exponential) procedures.

6. Apply *divide_and_conquer* induction to n. H will be induction hypothesis.

7. Satisfy the base and two step cases appropriately (i.e., substitute m for $\mathcal{M}$ in **(8)** and m for both $\mathcal{M}$ and the schematic matrix in **(9)**)

8. Verify each case by re-writing according to the properties of matrices until unification with the hypothesis, H, is possible.

9. Extract the logarithmic program. □

Example: logarithmic version of fib_3 via fib_{3_tup}

In §5.2.6 we described how, by following the GPPT, the IPOS automatically transforms the *Fibonacci* variant fib_3,

$$fib_3(n) \;=\; fib_3(n-1) + fib_3(n-3),$$

into a target fib_{3_tup},

$$fib_{3\_tup}(n) = \langle fib_3(n-1), fib_3(n-2), fib_3(n-3) \rangle,$$

thereby transforming an exponential procedure into a linear one. The linear version can, in turn, be automatically transformed into the logarithmic version by

appealing to the nth power of a $\Phi \times \Phi$ matrix m_{fib_3}. Since $\Phi = 3$ in the case of $fib_{3\_tup}(n)$ then, $m_{fib_3} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$,

such that:

$$\begin{aligned} \textbf{(i)} \quad fib_{3\_tup}(n) &= \langle fib_3(n-1), fib_3(n-2), fib_3(n-3) \rangle \\ &= \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^n fib_{3\_tup}(0)\,, \end{aligned}$$

i.e.,

$$\textbf{(i')} \quad fib_{3\_tup}(n) = m_{fib_3}(n) \times fib_{3\_tup}(0)$$

This corresponds to the *divide_and_conquer* induction hypothesis after existential elimination. The induction cases are then satisfied as follows:

- The base case is instantiated as follows:

$$\textbf{(ii)} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times fib_{3\_tup}(0) = fib_{3\_tup}(0)\,.$$

- The conclusion part of the first induction step is instantiated as follows:

$$\textbf{(iii)} \quad (m_{fib_3}(n))^2 \times fib_{3\_tup}(0) = fib_{3\_tup}(2n)\,.$$

Verification then proceeds by using matrix multiplication: **(iii)** is re-written to:

$$\textbf{(iii')} \quad m_{fib_3}(n) \times fib_{3\_tup}(0) = fib_{3\_tup}(n)\,,$$

which is fertilized with the induction hypothesis **(i')**.

- Similarly, the conclusion part of the second induction step is instantiated as follows:

$$\textbf{(iv)} \quad \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \times (m_{fib_3}(n))^2 \times fib_{3\_tup}(0) \;=\; fib_{3\_tup}(s(2n))\,,$$

and **(iv)** is then verified by the same reasoning as **(iii)**.

5.5.3 Making the Logarithmic Transformations more General

We now present the general form of the matrix required to transform linear procedures which have been obtained by transforming function definitions that fit the more general schema $EQ.1'$, repeated below:

$EQ.1'$ $f(x) \;\Leftarrow$ **if** $b(x)$ **then** $c(x)$
else $h(x, f_1(\delta^{a_1}(x)), f_2(\delta^{a_2}(x)), f_3(\delta^{a_3}(x)), ..., f_k(\delta^j(x)))$

The matrix is, as before, a $\Phi \times \Phi$ matrix. There are, however, two differences. The first is that the top row of the matrix is not restricted to containing 0's and 1's, but rather may contain any of the natural numbers depending on the nature of h. So if the recursive step of a function, foo, unpacks to the following:

$$foo(n) = m^1 \times (foo(n-2)) + m^2 \times (foo(n-4))\,,$$

then the first row of the corresponding 4×4 matrix will be $[\ 0 \quad m^1 \quad 0 \quad m^2\ ]$ (for a concrete example below). The second difference is that we must now allow for the fact that functions of the form $EQ.1'$ may have more than two subsidiary calls, and that the first of which may not correspond to $f(n-1)$, but $f(n-a_1)$ where a_1 is any natural number. This means that the first row of the $\Phi \times \Phi$

matrix will have a, possibly broken, block of natural numbers *starting* at position a_1 along the first row, and ending at position Φ (where $\Phi = j$).

The block of natural numbers may be broken since respective recursive arguments of the subsidiary calls need not necessarily be only one application of the common generator function out of step (as was the case for the fib_{tri} example of §5.2.6). The explicit definition of the linear tuple function, obtained by transforming a source program that computes an instance of $EQ.1'$, was given as follows (i.e., step 2 of the GPPT, §5.2.6):

$$(i)\ f_{tup}(n) \ = \ \langle f(\delta^1(n)), f(\delta^2(n)), f(\delta^3(n)), ..., f(\delta^{\Phi}(n))\rangle.$$

Using superscripts to represent the first row block of natural numbers thus $m^1 \_\_ \_\_ m^k$, and using subscripts as before to denote the position of each entry in a matrix row, we can express $f_{tup}(x)$ in terms of matrices thus:

$$f_{tup}(n) \ = \ \begin{bmatrix} 0_1 & 0_2 & 0_3 & . & 0_{a_1-1} & m^1_{a_1} & \_\_ & . & \_\_ & \_\_ & m^k_{a_\Phi} \\ 1_2 & 0 & 0 & . & 0 & 0 & 0 & . & 0 & 0 & 0 \\ 0_3 & 1 & 0 & . & 0 & 0 & 0 & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . & . & . \\ 0_\Phi & 0 & . & . & 0 & 0 & 0 & . & 1 & 0 & 0 \\ 0_\Phi & 0 & . & . & 0 & 0 & 0 & . & 0 & 1 & 0 \end{bmatrix}^n \times f_{tup}(0)\,.$$

The means by which logarithmic procedures, that evaluate matrices of the above form, are obtained from the linear procedures is precisely the same as before.

Example

Taking $fib_{\times\_3\_tri}$, of §5.2.6, as our example, the tuple

$$\langle fib_{\times\_3\_tri}(n-1), (2 \times fib_{\times\_3\_tri}(n-2)), (7 \times fib_{\times\_3\_tri}(n-3)), \\ fib_{\times\_3\_tri}(n-4), (4 \times fib_{\times\_3\_tri}(n-5))\rangle,$$

can be expressed in terms of a special matrix as follows:

$$fib_{\times\_3\_tri\_tup}(n) = \begin{bmatrix} 0 & 2 & 7 & 0 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} fib_{\times\_3\_tri}(\lambda^2\delta(n)) \\ fib_{\times\_3\_tri}(\lambda^3\delta(n)) \\ fib_{\times\_3\_tri}(\lambda^4\delta(n)) \\ fib_{\times\_3\_tri}(\lambda^5\delta(n)) \\ fib_{\times\_3\_tri}(\lambda^6\delta(n)) \end{pmatrix},$$

that is,

$$fib_{\times\_3\_tri\_tup}(n) = \begin{bmatrix} 0 & 2 & 7 & 0 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} fib_{\times\_3\_tri\_tup}(n-1),$$

Hence, we can express $fib_{\times\_3\_tri\_tup}$ in terms of the 5×5 matrix shown below:

$$H: \quad fib_{\times\_3\_tri\_tup}(n) = \begin{bmatrix} 0 & 2 & 7 & 0 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}^n \times fib_{\times\_3\_tri\_tup}(0),$$

On the application of *divide_and_conquer* induction H will become the induction hypothesis. Letting

$$m(n) = \begin{bmatrix} 0 & 2 & 7 & 0 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}^n,$$

then the induction cases of the *divide_and_conquer* schema are instantiated as follows:

base case: $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times fib_{\times\_3\_tri\_tup}(0) = fib_{\times\_3\_tri\_tup}(0)\,;$

1^{st} step: $m(n)^2 \times fib_{\times\_3\_tri\_tup}(0) = fib_{\times\_3\_tri\_tup}(2n)\,;$

2^{nd} step: $\left(\begin{bmatrix} 0 & 2 & 7 & 0 & 4 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times m(n)^2\right) \times fib_{\times\_3\_tri\_tup}(0) = fib_{\times\_3\_tri\_tup}(s(2n)).$

These are then verified through unfolding and fertilzation with H, as with the general case.

5.6 Summary

In this chapter we discussed program optimization through the original approach of (automatically) transforming inductive synthesis proofs. We described how exponential procedures are linearized by employing the proof tupling technique *combined with* source to target proof transformations. The latter involved abstracting information from the source proof in order to guide the target proof construction. In particular:

- The source proof is analysed for dependency information so as to construct a suitable explicit target tuple definition. *Alternatively*, we provided a simple heuristic which, by a direct mapping from source proof definitions, provides the requisite explicit definition without any dependency analyses.

- Information regarding introduction and elimination rules is abstracted from the source proof.

- Course of value induction schemata are transformed to stepwise induction schemata by using the former to provide the requisite subsidiary call dependencies required to instantiate the induction cases of the latter.

- The target proof is verified by mapping across the verification component of the source proof.

The IPOS approach to program transformation has different ramifications concerning correctness, search and the use of dependency information. Considering each, briefly, each in turn:

Correctness: As discussed in *Chapters 2* and *4*, the properties of the lower level OYSTER proof refinement system entail that, given a *complete* source specification then once a target proof has been completed then the meta-level transformation will be correctness guaranteed – simply because the target extract program satisfies the same complete specification as the source.

This entails that (i) we can employ heuristic techniques for the tupling transformations with the guarantee that, once terminated, the transformations are correct, and (ii) we do not have additionally to provide, or extend, any correctness criteria, or proof, each time we extend the range of programs to which the transformations are applicable.

Such considerations form one of the main motivating factors behind the IPOS design.

Reduced search space: With the more traditional program development systems which employ the fold/unfold technique, it is the *automation* of the lemma generating procedures and, *in particular*, the subsequent *folding* with the lemmas, that have proved, to date, difficult to automate - the difficulty increasing with the complexity of the folding operations required (which in turn depends on the complexity of the source algorithm together with the size of the class of transformations desired).

We described how target tuple definitions can be automatically generated by analysing source definitions. The problem of *folding* has been circumvented

within the proof transformations since due to the sequent calculus notation and the manner in which proofs are refined we need use only *unfolding*. Functions, and their definitions, produced by the *eureka* step are, in general, unfolded with the source induction branches in order to develop those of the target proof. So, in effect, recursive terms, corresponding to source proof induction terms, are eliminated from the target recursive branches, corresponding to the target proof induction branches, by *unfolding*.

The iterative application of unfolding, at least on available evidence, is far easier to automate. It has been done for the current system *and* within the context of automatic *proof plan* application through the automation of the *rippling out process* (*cf. Chapter 7*).

Abstracting dependencies: By exploiting information abstracted from the proof, specifically the account of the dependencies between facts involved in the computation, the tupling transformations have a reduced workload than those systems, reviewed in *Chapter 3*, that employ the tupling technique (Burstall & Darlington, 1977b; Darlington, 1981a; Chin, 1990). Although, in parallel with Chin's tupling analysis for programs in the HOPE$^+$ environment, a source inductive proof must satisfy tupling conditions and the actual tuple analysis hinges on dependency information, there is, however, no *additional* dependency graph analysis in order to control the tuple construction procedures since the source proof, and in particular the *unfolding* of any induction schemata, adequately exhibits all the dependency information required for tuple formation.

We also described an extension to the IPOS wherein linear procedures can be optimized to logarithmic procedures by using the method of *matrix multiplication* to represent a tuple as the application of the power of a special matrix object, and then replacing the *stepwise* induction employed in the source proof by a target *divide and conquer* induction.

Again, the transformation is guided by (sub)structures in the source proof. In particular, specific units in the source proof are accessed and, possibly after some

transformations, are used as witnesses for the base and step cases of the target schema.

The optimization of an exponential procedure to a linear procedure, through the automatic transformation of a source course_of_values inductive proof to a target stepwise inductive proof, followed by the optimization of a linear procedure to a logarithmic procedure, through the automatic transformation of a source stepwise inductive proof to a target divide_and_conquer inductive proof, illustrates how the choice of induction schema, and the subsequent satisfaction of its cases, is directly related to the form of recursion exhibited by the extract program. i.e., the choice of induction schema, and the subsequent satisfaction of its cases, is directly related to the form of recursion exhibited by the extract program.

Chapter 6

Conclusions and Further Work

6.1 Introduction

The research documented in this thesis has involved the following:

(i) An investigation into program transformation through proof transformation.

(ii) The (interactive) syntheses of different programs from common specifications.

(iii) A system for (automatically) adapting programs to special situations.

(iv) A system for (automatically) optimizing recursive programs.

We should like to emphasize again, as we did in *Chapter 1*, that the OMTS should be viewed as a research tool for the development of proof transformation methodologies. This is in a similar spirit to Darlington and Burstall's initial implementation of the NPL system, the primary function of which was as a research tool for the development of program transformation methodologies, (Burstall & Darlington, 1977b). The main difference is that the OMTS is the first excursion into investigating *both* (iii) and (iv) above through the transformation of (existence) proofs. As such, we do not pretend that the system is anything more than in it's early developmental stages. In particular, the problem of the *automation trade-off* effects the OMTS as it does practically all other existing transformation systems. Indeed, the trade-off would appear to be a fact of life as far as

program/proof transformation is concerned.[1] The global properties of the object-level proofs entail that the basic *design* of the OMTS will achieve considerable success in automating a large variety of recursive (extract) program transformations (*cf.* §6.2.1, item 7, and §6.2.2, item 3). The results achieved so far are encouraging. In the first half of this Chapter we summarize the main contributions afforded to both the proof and program transformation enterprises.

In the remainder of the chapter we suggest some interesting further avenues of research stemming from the discussions, and the descriptions of the implemented systems, contained within this thesis.

6.2 Contributions of Thesis

The *intuitive* motivation behind the design of the OMTS is that a proof will contain more information than the program which it specifies since a program need contain no more information than that required for execution. A proof, on the other hand, will contain the *thinking behind the program design.* The main contributions of this thesis stem from an investigation into how this extra information can be exploited for the task of program transformation.

We categorize these contributions into those that benefit, in some sense, the proofs as programs enterprise (§6.2.1), and those that mark an advance on the techniques and strategies reviewed in *Chapter 3* (§6.2.2).

6.2.1 Contributions w.r.t Proofs as Programs Paradigm

1. *Automated Proof Transformation*: This thesis research marks a successful start on the third component of the overall goal of tackling the demands on complexity, reliability, and quantity of software by tackling the problems of

[1]Indeed, considering any technique or strategy for automated reasoning, the search and control problems will, in general, increase considerably with the size of the problem domain one wishes the technique or strategy to encompass.

automated synthesis, verification, and *transformation* of programs: due to my thesis work advances have been made toward bringing the state of the transformation field in to line with the other two.

2. *Proof Transformations Allow for Easier Synthesis*: On empirical evidence alone, there appears to be an inverse relation between, on the one hand, the efficiency of the recursive process generated by an extract, and on the other, the complexity of the proof from which it was extracted.[2] This evidence has been gleaned from both the author's and the *Edinburgh Mathematical Reasoning Group*'s use of the NUPRL and OYSTER environment. In §2.2.8 we discussed the inverse relation using the two *Fibonacci* proofs as examples. Further evidence can be obtained by refering to (Madden, 1987b) where in the extracts corresponding to various synthesized sorting algorithms are compared with the syntactic density of the associated proofs.

   One practical contribution of a proof transformation system is, therefore, that it enables the synthesizer (human or mechanical) to construct short, elegant proofs, without clouding the design process with efficiency issues, and then to transform them into an opaque proof that yields an efficient program.

3. *Adapting Program Transformation Techniques*: We have successfully, and for the first time, adapted a range of program transformation techniques to the proofs as program paradigm:[3]

   - partial evaluation;
   - tupling; and
   - fold/unfold.

---

[2]This is despite the fact that human theorem provers are usually trained to find short, elegant proofs rather than long opaque ones.

[3]Where, recalling §5.4.3 in the course of adapting the fold/unfold technique, we actually removed the fold requirement – *cf.* §6.2.2, item (iii)).

The transformations exhibit the desirable criteria of correctness and automatability. Furthermore, the OMTS is equipped with numerous control mechanisms for guiding the target construction through the transformation search space (§6.2.2).

Apart from the specialization application documented in (Goad, 1980b; Goad, 1980a), the OMTS offers the only *working* example of an (automatic) transformation system that optimizes programs through proof transformations.

4. *Transforming Inductions*: Although C.A.Goad has previously investigated program specialization through pruning operations that simplify synthesis proofs, this thesis offers the first description of a working system that (in addition to specialization) optimizes *recursive programs* by transforming the corresponding *inductive proofs*. I.e., the OMTS is the first program transformation system which directly exploits the duality between mathematical induction, as employed during constructive syntheses, and recursion.

5. *Reacting to Changing Specifications/Requirements*: In *Chapter 1, section 2*, we drew attention to the difficulty of reacting to changing specifications. Regarding traditional programming, this difficulty is compounded by the fact that modification of the source code, according to a modification in the source specification, relies almost exclusively on the programmer's understanding of the code he has to modify.

   Although still a problem when programs are defined through proofs, the ability to make local changes to a proof structure, in accordance with changes in the proof specification, is made easier by the fact that proofs represent much more of a design record of the program being synthesized (i.e., the procedural decisions and commitments are rendered explicit within the proof tree).

   Proof transformations, in general, reduce the problem even further by providing a way of making those local changes and propagating them throughout the proof since we can identify places where additional theorem proving is

required in order to meet the changed specification or verify the modified program.

5(i) *Regarding Specialization*: the simplification of a source synthesis proof tree, by performing proof pruning transformations according to the initial partial instantiation of the source extract program's parameters, provides a good example of how proof transformation can assist in modifying a program's internal structure in accordance with an initial modification of its specification.

5(ii) *Regarding the Optimization of Recursive Programs*: modifying the source induction schema provides a good example of how proofs enable us to react to changes other than modifications of the program specification, i.e., reacting to changes in a program's internal structure. Since inductive proofs follow the same general strategy *and* since we can identify particular schemas with kinds of recursion then we can formulate general strategies for:

- optimizing the source program's recursion schema by transforming the source proof induction schema; and
- subsequently, verifying that the modified source recursion schema – or target recursion schema – preserves the source input/output relation by verifying the instantiated target schema with respect to the source specification.

6. *An Indirect Solution to Requirements Capture*: Recall that Requirements Capture is the problem of forming the specification in the first place (§2.2.9). That is, forming specifications which, upon refinement, provide algorithmically efficient programs. So, related to point 2 above, if we have suitable optimization systems the need to address the problem of requirements capture becomes less urgent. This is because as long as the specification satisfies the *minimum* conditions of stating the desired program input-output relation correctly, then this will yield a proof/program which can *then* be suitably

optimized. We would not, however, pretend that this marks a preferable approach to addressing the problem of refinement capture head on.

7. *Generality (and expectations thereof)*: Although the actual implementation should be regarded as embryonic, the success that it has achieved, together with the property of inductive proofs that they share a common structure/strategy, suggest that a far broader corpus of recursive programs, than those covered in this thesis, can be optimized using the same proof transformation system design. Further backing for this prediction is afforded by the success of the CLAM proof-planner, much of which is to be attributed to the common structure and theorem proving strategies of inductive proofs, and the ability to construct typical proof-plans (very much akin to the OMTS rule-tree abstractions) from example proofs.

6.2.2 Contributions w.r.t. Advances on Existing Transformation Systems

We first summarize the benefits and novel features of the OMTS when compared to traditional program transformation systems, and then we summarize the advances made by the OPSS on Goad's original program specialization by proof transformation system.

Regarding Program Transformation

Tupling has been used in previous program transformation systems (Cohen, 1983; Darlington, 1981b; Chin, 1990). This thesis is, however, the first attempt to transpose the (automatic) tupling technique to the *proofs as programs* paradigm. On the available evidence of the OMTS performance, the use of proof tupling *combined with* induction schema transformation is a successful approach to linearizing exponential source programs. We itemize the main benefits of our approach as follows:

1. *Correctness*: As discussed in *Chapters 2* and *4*, the properties of the lower level OYSTER proof refinement system entail that, given a *complete* source specification then once a target proof has been completed then the meta-level transformation will be correctness guaranteed – simply because the target extract program satisfies the same complete specification as the source. This entails that:

   (i) we can employ heuristic techniques for the tupling transformations with the guarantee that, once terminated, the transformations are correct; and

   (ii) we do not have to additionally provide, or extend, any correctness criteria, or proof, each time we extend the range of programs to which the transformations are applicable.

2. *Reduced search space and Reducing Search (or Increasing Control)*: The most general factor that aids in guiding the transformation of a source proof rule-tree through the transformation space is *meta-level control.* That is, transformation tactics are selected according to whether or not certain syntactic properties of the source rule-tree (and thereby the source proof) are violated. Thus, for example, to specialize a source proof, transformation tactics can be selected by first observing that the pre-conditions for normalization are present, and then by observing that the post-conditions of normalization make the pre-conditions for dependency pruning true. Similarily, the post-conditions for tupling transformations will match with the pre-conditions for the matrix multiplication transformations discussed, as an extension to the current implementation of the OMTS, in §6.3

   However, meta-level control is by no means a novel feature of the OMTS. Furthermore, although the meta-level design has been fully implemented, the pay-offs of meta-level control will be better realized when the OMTS has access to a greater set of transformation tactics which can be "dove-tailed" together in accordance with their pre- and post-conditions.

In all, there are five main factors that contribute, in *original* ways, toward either reducing the transformation search space, or controlling the search within that space:

(i) the means by which dependency information is abstracted and exploited for the purposes of transformation;

(ii) the tupling heuristic;

(iii) the removal of the fold step from the unfold/fold strategy;

(iv) the exploitation of similarities, or analogies, between the source proof constructs and those required to construct the target proof; and

(v) the exploitation of the general strategy for (inductive) theorem proving.

We shall consider (i), (ii) and (iii) in this section, and (iv) and (v) will be summarized in the subsequent section.

(i) *Abstracting dependencies*: By exploiting information abstracted from the proof, specifically the account of the dependencies between facts involved in the computation, the proof tupling transformations circumvent the need for the computationally expensive construction, and subsequent analyses, of dependency graphs required of previous systems that employ the tupling technique.

Although, in parallel with Chin's tupling analysis for programs in the HOPE$^+$ environment, a source inductive proof must satisfy tupling conditions and the actual tuple analysis hinges on dependency information, there is no *additional* dependency graph analysis required to control the tuple construction procedures. This is because the source proof, and in particular the *unfolding* of any induction schemata, adequately exhibits all the dependency information required for tuple formation. This is true of both the OMTS tuple construction procedures (T1 and T2, §5.2.4), where the former is guaranteed to produce a tuple for a specified large class of functions ($EQ.1'$, §5.2.6), and the latter, although not guaranteed to produce a tuple, cuts down even further on

the amount of analysis required to produce a tuple in those instances where it succeeds (see next item (ii))

A nice illustration of the fact that the information required by tupling transformations performed on program code is (explicitly) present within the synthesis proofs is that Chin's automatic analysis of unfolding cuts, followed by unification is very similar to the process of rippling out followed by fertilization during synthesis.

(ii) *Heuristic control*: We provided a simple, effective, and unique heuristic, T2, for automatic tupling – in addition to the non-heuristic procedure T1 – which, by a direct mapping from source proof definitions, provides the requisite explicit definition without any dependency analyses. T2 circumvents much of the computational analysis required by T1 (although T1, in turn, involves considerably less computational analysis than in previous tupling transformations on program code, *cf.* (i) above). Although not guaranteed to produce a suitable tuple, T2 will do so for most simple functions and succeeds in some cases where T1 fails.

(iii) *Eliminating the fold in fold/unfold*: The *proof* transformations allow us to remove the *fold* step from the unfold/fold strategy, hence removing the associated control problems of deciding *when* to fold. The resulting strategy, consisting mainly of controlled unfolding, also has a reduced search space and, upon available evidence, is easier to automate.

The iterative application of unfolding, at least on available evidence, is far easier to automate. It has been done for the current system *and* within the context of automatic *proof plan* application through the automation of the rippling-out process.

Regarding Proof Transformation

We wish to emphasize three contributions toward the limited field of program transformation through proof transformation. The first concerns the correctness

of the specialization (proof) transformations, and the second concerns the OPSS induction grounding transformations. Finally, we discuss the benefits of exploiting the global property of inductive proofs that they pertain to a common structure (or exhibit a common strategy).

1. *Ensuring the Correctness of Proof Transformations*

In addition to the general criteria of correctness for the source to target proof transformations, the presence of a (complete) program specification presents us with a novel state of affairs within the program transformation enterprise: a correctness guarantee for the source to target transformation of a program's functionality.

Goad is forced to limit his pruning transformations to those that (can be proven to) preserve the validity – but *not* necessarily the equivalence – of an algorithm for the specification embodied in the root node of the proof describing the algorithm. Such a state of affairs demands a fairly restrictive class of input program: those that satisfy a specification that is full but not complete. In this way Goad can alter the functionality – or more precisely, the input/output *range* – of such a specified program without altering the specification itself.

Regarding the OPSS methodology of rule tree (proof-plan) transformation, by virtue of the presence of a specification (and corresponding proof) at each stage of the specialization process, we can guarantee the correctness of any target program for both the following cases:

(i) specializations where the source specification is preserved – as in dependency pruning; and

(ii) specializations where the source specification is transformed – as in induction grounding.

In the case of (i), we circumvent any need to provide "validity proofs". Regarding (ii), there is no possibility of providing such "validity proofs" since such transformations presuppose that we actually modify the source specification, and then

ensure the correctness of the target program by extracting it from a proof of that modified specification.

2. *Induction Grounding*

The OPSS includes a third pruning mechanism, induction grounding, which is designed to specialize recursive behaviour through pruning source sub-proof trees associated with the application of mathematical induction. Induction grounding serves as an extension to the pruning transformations described and implemented by Goad, and provides an example of where extensions to the pruning transformations can be afforded a correctness guarantee regardless of whether or not the source and target programs satisfy the same specification.

Induction grounding does not have the same, rather stringent, pre-conditions as does dependency pruning, the only requirement being that the source proof is an inductive proof (yields a recursive program). Consequently, induction grounding has a wider field of application than the other pruning transformations.

Finally, the presence of a complete target proof means that we have the choice as to whether to go on to perform further *proof* transformations, an example being the OPSS normalization, and possibly dependency pruning, of a recursive program followed by the OMTS optimization of it's recursion schema.

3. *Exploiting Typical Proof Strategy to Reduce Search in Proof Transformation*

Both Goad and Pfenning, (Pfenning, 1988), use, or suggest using, the *proofs as programs* paradigm in order to exploit the properties of proofs so as to guide the transformation of (extract) programs. However, neither mentions exploiting the global property of inductive existence proofs, that the majority pertain to a common structure or shape (§2.2.4), in order both to increase expectations of generality of, and as a guide for, the source to target transformation of inductive proofs.

Since the majority of inductive existence proofs will exhibit a common theorem proving strategy (depicted in **fig.** 2–1, *Chapter 2*) then, in all probability, the

application of induction (including the witnessing of the induction cases), and the subsequent verification steps, will overlap considerably between the source and target proofs. By using the typical inductive proof strategy as a skeleton proof plan, we can guide the development of the target instance of such a plan.

4. *Exploiting Analogy to Further Reduce Search in Proof Transformation*

The source and target proofs will share a greater degree of similarity than that reflected merely by the general inductive proof strategy (3 above) by virtue of the fact that both are proving the same specification. Hence the source proof provides a more accurate guide – or a closer analogy – regarding the witnessing of the induction cases and any lemmas employed to complete the target construction. So, in practice, the source proof will generally provide more detail for guiding the target construction than does the typical inductive proof strategy, where as the latter renders the *design* of the inductive proof transformations general to the majority of inductive proofs.

6.3 Further Work Proposals: Efficient CLAM Proof-Plans

The OYSTER search space is very large, even by theorem proving standards. There are hundreds of rules of inference, many of which have an infinite branching rate. So careful search is very important if a combinatorial explosion is to be avoided.[4]

The CLAM proof-plans provide a technique which is used to guide automatic inference through such a search space in order to avoid a combinatorial explosion. The control and inference information – formulae, axioms and inference patterns (or patterns of reasoning) – employed by humans when proving theorems

[4]A large proportion of the OYSTER search space consists of sub-proofs that various expressions are well typed.

in mathematics have been represented, as proof-plans, in a formal meta-language (or meta-theory) (Bundy *et al*, 1990b; Bundy *et al*, 1991).

The assertions of the meta-theory describe properties of the object theory. Hence in the meta-theory we can formally reason about solutions to object-level problems and different methods for obtaining these solutions. Such an approach has already been successfully applied to the domains of equation solving (PRESS), (Bundy & Welham, 1981), and is now being extended to the domain of (automatic) proof synthesis and verification (Bundy, 1988b; Bundy *et al*, 1991).

The formalization of the CLAM meta-theory for the OYSTER proof refinement system has enabled deduction and learning theories to be applied to the meta-theory itself. Lower level tactics used to guide object level inference have been combined in various ways such that proof-plans can be represented, still within an overall uniform logic framework. The proof-plans can be generalized from sample solutions and are themselves amenable to manipulation.

The further work proposals regarding the CLAM proof-plans consists of the following (Madden, 1991):

1. Heuristics relating the efficiency of algorithms with the methods employed in the synthesis proofs will be incorporated into the proof-plan representation. Hence, extended proof-plan techniques will be used to guide the synthesis of more efficient algorithms.

2. Assuming that the proof-plan technique is extended, as in **1** above, there is still no guarantee that searching for the most efficient structures will be computationally less costly than transforming an inefficient proof found with the unextended proof-plan technique. So the worth of incorporating the heuristics, referred to in 1 above can be investigated by comparing their performance with that of the OMTS.

3. The third proposal involves augmenting the existing CLAM methods with "super-methods" that take complete OMTS rule-trees as input, and output

rule-trees that, when applied at the OYSTER object-level, yield more efficient (extract) programs.

4. The final proposal is to interface the CLAM system with the OMTS system, such that proof-plans yielding *efficient* programs are automatically constructed from a program specification. The proposal assumes certain advances upon the current performances of both the OMTS and CLAM systems.

6.3.1 Extending the proof-plan technique

The first stage of the proposed further research involves extending the proof-plan technique such that the efficiency of the recursive structures extracted from proofs plays an important role in the proof-plan search strategy.

At present the proof-plan technique certainly circumvents the vast search problem and is likely to find *some* proof which satisfies the desired input/output specification. However, this proof will, more often than not, be the simplest in structure and will not necessarily yield the best algorithm in terms of efficiency.

Recall, §2.2.8, that there is an interesting inverse relation between the complexity of proofs and that of the algorithms which they synthesize such that the more efficient the algorithm the more complex the proof from which it is extracted. So we would wish the search strategy to be tailored such that the simplest *proof* is not the one it finds first, but rather that which yields an efficient algorithm.

This tailoring could be achieved by setting an *induction schema precedence* on the proof-plan technique. That is, since the efficiency of recursive programs is directly related to the type of induction employed in their synthesis then the search for proofs would, at least partially, be guided by a precedence ordering on which particular induction schema is employed in the proof.

Of course, there are further efficiency factors involved with how the chosen induction schema is applied, notably which variable, among the possible alternatives, is chosen as the induction variable (Aubin, 1975; Aubin, 1976; Boyer & Moore, 1979). These could also be incorporated into the overall proof-plan search strategy.

Building such efficiency considerations into the way CLAM selects tactics will clearly increase the proof-plan search space. This runs contrary to one of the motivations behind the OMTS: that it enables one to separate, on the one hand, the process of synthesizing a program from a specification, and on the other, synthesizing a program that *efficiently* computes the specified output from the input. The proposal of §6.3.3 attempts to combine the automatic synthesis of CLAM with the automatic optimization of the OMTS whilst keeping the two process distinct.

6.3.2 Assessing the Performance of an Extended Proof-Planner

Assuming that the proof-plan technique is extended as described above then there is still no guarantee that searching for the most efficient structures will be computationally less costly than using the proof transformation system, discussed in *section 4*, to transform an inefficient proof found with the unextended proof-plan technique. The extended proof-plan technique may involve forced backtracking such that the proof-planner is *not* geared to come up with the simpler proofs, or at least not the most complex refinement trees, first. Although this may well produce more efficient extract programs, it may also, however, cause additional overhead. So the second stage of the proposal involves an empirical investigation: the search and control problems of automatically constructing an efficient program using the extended CLAM proof planner are compared with those associated with transforming the inefficient program output by the non-extended proof planner.

6.3.3 A Combined System: The CLAM–OMTS Efficient Proof Planner

Regarding the OMTS, the rule-trees, which are tantamount to proof-plans (§2.3.3), are not produced from such a recursive application of methods to a single input formulae (theorem or specification). Rather a target rule-tree is produced from a

source rule-tree, the latter being abstracted from the source proof itself (§5.3.3). So whereas the CLAM proof-planner takes a theorem as input and outputs a completed proof-plan, the OMTS both takes as input and produces as output a complete rule-tree (proof-plan).

The CLAM proof-plans differ slightly from the OMTS rule-trees in that, regarding the latter, whenever certain object-level rules are applied – such as the $decide(X)$ and $seq(Y)$ refinements – any newly introduced hypotheses are *explicitly* labelled and recorded within the rule-tree. Similarly, certain hypotheses discharged during a proof are also recorded within the OMTS rule-trees, but not within the CLAM proof-plans. This is an essential ingredient for the OMTS transformations that exploit dependency information. Such (dependency) information is invisible within the CLAM proof-plans since it is not required for the purposes of theorem proving and/or synthesis. However, it would be simple to matter to augment the CLAM proof-plans such that they can be utilized, as rule-trees, by the OMTS. This would be done by ensuring the explicit recording of certain hypotheses made and/or discharged in the course of constructing the tactic methods. Henceforth, we shall refer to the OMTS constructs as either proof plans or rule-trees depending on context.

In practice, when transforming a proof the OMTS takes the actual proof, rather than it's corresponding rule-tree/proof-plan as input. However, the system design is fairly modular such that *if* we were given a rule-tree/proof-plan to start with, the OMTS could proceed with this by simply omiting the initial rule-tree abstraction phase. We shall hence speak of the input (output) of the OMTS being either a proof or a proof plan depending on context.

Planning Existential Proofs

The CLAM system is at present more successful at planning non-existential inductive proofs. This is because proofs of existential specifications – i.e., synthesis proofs – present more complex control problems, due mainly to the proliferation of choices at the witnessing steps.

The reason that the OMTS is capable of automatically producing a proof-plan for the example existential proofs (*Chapters* 5 and *6*) is that it exploits the control and inference information in a source proof-plan to guide it through the tricky "eureka" steps required in the target proof-plan construction.

However, ongoing research is aimed at increasing CLAM's capability of performing automatic synthesis, as well as automatic theorem proving *per se*. Recent successes include CLAM's ability to automatically find a proof of the existence of prime factorization which, not only verifies a given algorithm for prime factorization, but also synthesizes this algorithm from an existentially quantified specification (Bundy *et al*, 1991). This is a significant landmark, since such automatic *synthesis* is beyond the scope of the "state-of-the-art" Boyer-Moore theorem prover with which CLAM is often compared (Boyer & Moore, 1988).

Interfacing CLAM *with the* OMTS

Assuming that the ability of CLAM to automatically construct and apply proof-plans for synthesis proofs is increased, along with the ability of the OMTS to transform a greater diversity of proofs, then a completely automatic system for synthesizing *efficient* programs from their specifications could be achieved by interfacing the CLAM, OMTS and (object-level) OYSTER systems.

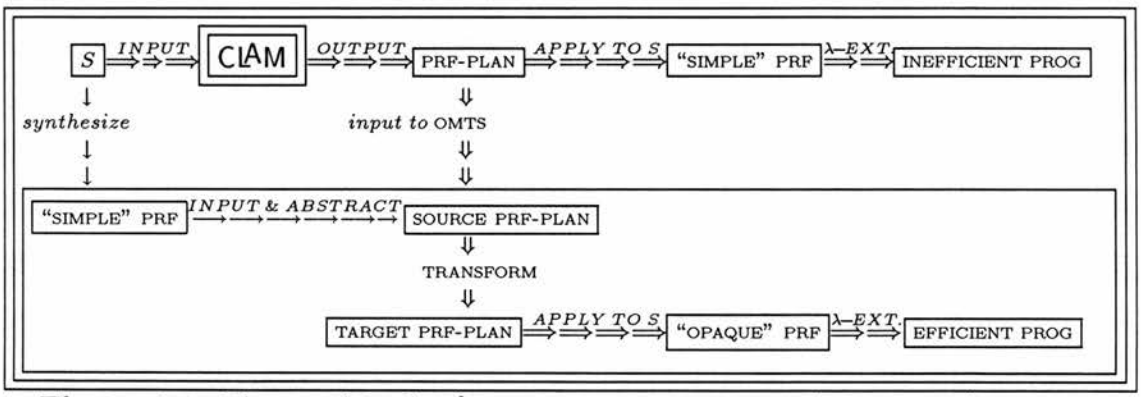


Figure 6-1: The combined CLAM–OMTS automatic synthesis–transformation process.

The basic design idea of the combined system is illustrated in **fig.** 6–1 above. Note that the diagram depicts the alternative means of using the OMTS, either

by entering a manually synthesized proof, depicted by thin bodied arrows of the form $\longrightarrow$, or by having a proof-plan entered directly from CLAM. All thick bodied arrows of the form $\Longrightarrow$ denote an automatic process.

Properties of Combined System (with reference to fig. 6-1 above)

- The combined system will operate completely automatically. The input will be a (complete) specification for the desired program, and the output will be an efficient procedure for computing that specification.

- However, the user will still has the option of hand feeding a synthesized source proof into the OMTS sub-system.

- Due to the fact that the OMTS rule-trees are virtually the same as the CLAM proof-plans, there will be minimal overhead caused by interfacing the two systems. On the contrary, overhead may well be reduced by interfacing the systems, as depicted in **fig.** 6–1 since we circumvent the need to abstract a rule-tree (proof-plan) from the source proof in the first place (the output of the CLAM sub-system doing the job for us).

- The correctness of the final efficient program is ensured by the presence of a target specification (the source-to-target correctness being ensured by virtue of the identicality of the source and target specifications).

- *Modularity*: Unlike the proposal outlined in §6.3.1 above, we avoid the need to cloud the CLAM design process with efficiency considerations. I.e., the CLAM sub-system is responsible for automatically synthesizing (a proof-plan that yields) a proof that computes the specified input-output, and the OMTS sub-system is responsible for automatically increasing the efficiency with which the output is computed from the input.

6.3.4 Designing CLAM "Super-methods"

The following properties are true of the OMTS source to target proof transformations (such as the pruning transformations, induction schema transformations and (proof) tupling transformations):

- There are formal pre-conditions required of a source proof in order that the transformation is likely to succeed.

- There are formal post-conditions such that we can predict the likely outcome of applying a transformation, without actually applying it.

So, in effect, the OMTS source to target proof transformations function rather like "super-methods" that take proof-plans as input rather than the smaller OYSTER tactics. For example, tupling transformations on a source *course_of_values* proof could be informally expressed as the following "super-method":

input slot: source proof-plan;

pre-conditions: For example:

- an application of course_of_values induction;
- there exist two or more induction terms which share some *common induction variable(s)* in a function definition (i.e., tupling pre-conditions);

output slot: target proof-plan;

post-conditions: A sequenced application of stepwise induction on a tuple.

The post-condition slot could be fleshed out with more information such as the target tuple must contain an equal number of components as there are subsidiary recursive calls in the source definition (*cf.* §5.2.3).

Similarly, the dependency pruning transformations could be informally expressed as the following "super-method".

input slot: source proof-plan;

preconditions: preconditions for dependency pruning (i.e., the presence of a case analyses, C, with a redundant case condition, *cf.* §4.2.4);

output slot: target proof-plan;

postconditions: postconditions for dependency pruning (i.e., the presence of the non-redundant case condition, with the corresponding proof branch, in place of C).

As well as using theorem proving heuristics, such as rippling out, the process of applying a super-method would be guided by precisely the same kind of information used to guide the OMTS transformations:

1. Information concerning the relative efficiency of the dual recursions associated with the various induction schemas can be used to replace the source induction schema with a target schema that induces more efficient recursive behaviour in the extract program.

2. Information extracted from the source proof, for example:

   (a) dependency information can be used to "patch" the target plan by introducing an existential witness at the target induction step case that combines sub-computations associated with the source induction; and

   (b) since many of the facts, or propositions, in the target proof will correspond to facts in the source proof then we can map across accordingly the lemmas/definitions required for the target verification.

3. Program transformation techniques (such as tupling) can be used for the "merging" of the source sub-computations.

The main work required in order that such transformation "super-methods" could be incorporated into the CLAM system would be the formalization of the

"super-methods" in CLAM's meta-logic (so the domain of discourse of the meta-logic would, as before, include logical expressions and tactics for manipulating these expressions, but would in addition include proof-plans and some kind of operators for manipulating the tactics within). Since, as we noted earlier, proof-plans just are large tactics that produce complete proofs then such formalization of the transformation "super-methods" should not be problematic.

References

Aubin, R. (1975). Some generalization heuristics in proofs by induction. In Huet, G. and Kahn, G., (eds.), *Actes du Colloque Construction: Amelioration et verification de Programmes.* Institut de recherche d'informatique et d'automatique.

Aubin, R. (1976). *Mechanizing Structural Induction.* Unpublished Ph.D. thesis, University of Edinburgh.

Bibel, W. and Hörning, K.M. (1984). Lops — a system based on a strategical approach to program synthesis. In Biermann, A., Guiho, G. and Kodratoff, Y., (eds.), *Automatic Program Construction Techniques*, pages 69–90. MacMillan.

Bjerner, B. (1989). *Time Complexity of Programs in Type Theory.* Unpublished Ph.D. thesis, University of Göteborg.

Boyer, R.S. and Moore, J.S. (1979). *A Computational Logic.* Academic Press, ACM monograph series.

Boyer, R.S. and Moore, J.S. (1988). *A Computational Logic Handbook.* Academic Press, Perspectives in Computing, Vol 23.

Bruynooghe, M., de Schreye, D. and Krekels, B. (1989). Compiling control. *Journal of Logic Programming*, pages 135–162.

Bundy, A. and Welham, B. (1981). Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212. Also available from Edinburgh as Research Paper 121.

Bundy, A. (1988a). A broader interpretation of logic in logic programming. Research Paper 388, Dept. of Artificial Intelligence, Edinburgh, To appear in the Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.

Bundy, A. (1988b). The use of explicit plans to guide inductive proofs. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag. Longer version available from Edinburgh as Research Paper No. 349.

Bundy, A., Sannella, D., Giunchiglia, F., Harmelen, F. Van, Hesketh, J., Madden, P., Smaill, A., Stevens, A. and Wallen, L. (1988). Proving properties of logic programs: A progress report. In *1988 Alvey Conference*, pages 131–133.

Bundy, A., van Harmelen, F., Smaill, A. and Ireland, A. (1990a). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449.

Bundy, A., van Harmelen. F., Horn, C. and Smaill, A. (1990b). The Oyster-Clam system. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning.* In press. Earlier version available from Edinburgh as Research Paper No 413.

Burstall, R. and Darlington, J. (1977a). A transformation system for developing recursive programs. *Journal of Assoc. Compt. Mach.*, 21(1):44–67.

Burstall, R.M. and Darlington, J. (1977b). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67.

Carbonell, J.G. (1983). Learning by analogy: Forming and generalizing plans from past experience. In Michalski, R.S., Carbonell, J.G. and Mitchell, T.M., (eds.), *Machine Learning: An Artificial Intelligence Approach*, pages 137–160. Tioga Press.

Chin, W. N. (1990). *Automatic Methods for Program Transformation.* Unpublished Ph.D. thesis, University of London (Imperial College).

Clark, K.L. and Darlington, J. (1980). Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65.

Clark, K. L. (1977). The synthesis and verification of logic programs. Research report (ccd), Imperial College.

Clark, K.L. (December 1979). Predicate logic as a computational formalism. Report 79/59, Department of Computing, Imperial College, London.

Cohen, N. H. (1983). Eliminating redundant recursive calls. volume 5 No. 3, pages 265–299.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Curry, H.B. and Feys, R. (1958). *Combinatory Logic.* North-Holland.

Darlington, J. (August 1981). An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46.

Darlington, J. (1981b). An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46.

Darlington, J. (1989). A functional programming environment supporting execution, partial evaluation and transformation. In *PARLE 1989*, pages 286–305, Eindhoven, Netherlands.

De Schreye, D. and Bruynooghe, M. (1989). On the tranformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, (7):125–154.

Feather, M. S. (1979a). *A System For Developing Programs by Transformations.* Unpublished Ph.D. thesis, University of Edinburgh.

Feather, M.S. (1979b). *A System for Developing Programs by Transformation.* Unpublished Ph.D. thesis, University of Edinburgh.

Goad, C. A. (1980a). Computational uses of the manipulation of formal proofs. Technical report, Stanford University, STAN-CS-80-819.

Goad, C. A. (1980b). Proofs as descriptions of computation. In *Lecture Notes in Computer Science.* Academic Press.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematic a und verwandter Systeme I. *Monatsh. Math. Phys.*, 38:173–98. English translation in (Heijenoort 67).

Grant, P. W. and Zhang, J. (1988). Heuristics for the automatic transformation of Prolog programs. In *UK IT 88 Conference Publication*, pages 135–139. UK IT 88 Conference (sponsors: SERC, DTI, MOD).

Green, I. (1991). Flexible program transformation. Technical report, University of Cambridge.

Gregory, S. (1980). Towards the compilation of annotated logic programs. Technical Report CCD no 80/16, Imperial College, London.

Harper, R., Honsell, F. and Plotkin, G. (1987). A framework for defining logics. In *Proc. of the Second Symposium on Logic in Computer Science.*

Heijenoort, J van. (1967). *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Mass.

Hogger, C.J. (1980). *Derivation of Logic Programs.* Unpublished Ph.D. thesis, Imperial College.

Hogger, C.J. (April 1981). Derivation of logic programs. *JACM*, 28(2):372–392.

Horn, C. and Smaill, A. (1990). Theorem proving with oyster. Research Paper 505, Dept. of Artificial Intelligence, Edinburgh, To appear in Procs IMA Unified Computation Laboratory, Stirling.

Howard, W.A. (1980). The formulae-as-types notion of construction. In Seldin, J.P. and Hindley, J.R., (eds.), *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press.

Huet, G. and Lang, B. (1978). Proving and applying program transformation expressed with second order patterns. *Acta Informatica*, 11:31–55.

Kleene, S. C. (1945). On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124.

Kott, L. (1978). About a transformation system: A theoretical study. In *Proc. of 3rd Symposium on Programming*, pages 232–267, Paris. 3rd Symposium on Programming.

Kowalski, R. (1979). Algorithm = Logic +Control. *Communications of ACM*, 22:424–436.

Kreisel, G. (1959). Interpretation of analysis by means of constructive functionals of finite type. In *Constructivity of Mathematics*, pages 101–128. North-Holland.

Kreisel, G. (1975). Some uses of proof theory for finding computer programs. *Colloques Internationaux du Centre National de la Researche Scientifique*, 249:123–134.

Kreisel, G. (1977). Some uses of proof theory for finding computer programs. *Recueil Des Travaux de L'Institut Mathematique*, 2:63–72.

Madden, P. (1985). Learning by analogical transformation. Unpublished M.Sc. thesis, Department of Cognitive Science, University of Sussex.

Madden, P. (1986). The application of analogy to program optimization. Research Paper 54, Dept. of Artificial Intelligence, University of Edinburgh.

Madden, P. (September 1987). The application of analogy to program optimization. Discussion paper 51, Dept. of Artificial Intelligence, Edinburgh.

Madden, P. (1987b). A nuprl synthesis of several sorting algorithms: Towards an automatic program transformation system. Research Paper 356, Dept. of Artificial Intelligence, Edinburgh.

Madden, P. (June 1988). Automatic program optimization via the transformation of Nuprl synthesis proofs. Research paper 392, Dept. of Artificial Intelligence, Edinburgh, Also in Proceedings of the 1988 Alvey Technical Conference.

Madden, P. (1988b). Automatic program optimization via the transformation of Nuprl synthesis proofs. In Clarke, L., (ed.), *Proceedings of the 1988 Alvey Technical Conference.* The Alvey Directorate.

Madden, P. (1988c). The Lops approach to program synthesis. Research Paper 406, Dept. of Artificial Intelligence, Edinburgh.

Madden, P. (1988d). The specialization of constructive existence proofs. Research Paper 406, Dept. of Artificial Intelligence, Edinburgh.

Madden, P. (1989). The specialization and transformation of constructive existence proofs. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

Madden, P. (1991). Automatic proof plan optimization. Research Paper: submitted to CADE-91, Dept. of Artificial Intelligence, University of Edinburgh.

Manna, Z. and Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121.

Martin-Löf, Per. (August 1979). Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover. Published by North Holland, Amsterdam. 1982.

Martin-Löf, Per. (1984). *Intuitionistic Type Theory.* Bibliopolis, Naples, Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

Michie, D. (1968). Memo functions and machine learning. *Nature*, 218:19–22.

Miller, D. and Nadathur, G. (1988). An overview of λProlog. In Bowen, R. K. & Kowalski, (ed.), *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press.

Pettorossi, A. (1984). A powerfull strategy for deriving programs by transformation. In *ACM Lisp and Functional Programming Conference*, pages 405–426.

Pfenning, F. (1988). Program development through proof transformation. Research Paper ERGO-88-047, Department of Computer Science, Carnegie Mellon University.

Prawitz, D. (1965). *Natural Deduction.* Almquist and Wksell, Stockholm.

Scherlis, W. L. (1980). *Expression Procedures and Program Derivations.* Unpublished Ph.D. thesis, Stanford University.

Smaill, A. (1990). Divide and conquer for the natural numbers. Blue Book Working Paper 512, Dept. of Artificial Intelligence, University of Edinburgh.

Stevens, A. (1988). A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In Kodratoff, Y., (ed.), *The Proceedings of ECAI-88*, pages 565–570. European Conference on Artificial Intelligence. Also available from Edinburgh as Research Paper No. 360.

Szabo, M.E., (ed.). (1969). *The Collected Papers of Gerhard Gentzen.* North Holland.

Tamaki, H and Sato, T. (1984). A transformation system for logic programs that preserves equivalence. Technical Report TR-018, ICOT.

van Harmelen, F. and Bundy, A. (1988). Explanation-based generalization = partial evaluation. *Artificial Intelligence*, 36(3):401–412. Also available as Edinburgh DAI Research Paper 347.

van Harmelen, F. (1989). The CLAM proof planner, user manual and programmer manual. Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh.

Wainer, S. S. (1990). Logical and recursive complexity. Technical Report 31/90 (Preprint Series), Center for Theoretical Computer Science, University of Leeds.

Appendix A

Abstract Transformation: Using Inductions to Construct Inductions

Course_of_values induction, *stepwise* induction and *divide_and_conquer* induction are all constructively inter-derivable and hence provably equivalent. Showing how a target induction schema can be constructively derived from a source schema, independent of any particular proofs, will afford us with a general theoretical story behind employing the object-level proof refinement system as an interactive transformation system.[1] For example, if we can constructively derive a *stepwise* induction schema from a *course_of_values* induction schema *then* we can derive a *course_of_values* recursion schema from a stepwise recursion schema (i.e., the corresponding extract program constructs).

The proof of equivalence means that as far as provability goes one is just as good as the other (though as far as usability goes they may be quite different). In other words, if we have a verified source proof employing course-of-values induction then there is a logically equivalent proof employing stepwise induction. The same applies to *stepwise* and *divide_and_conquer* induction, and therefore to *course_of_values* and *divide_and_conquer* induction (although we do not provide a proof of equivalence regarding *divide_and_conquer* induction. For this readers should consult (Smaill, 1990)).

---

[1]The author is indebted to Dr. A. Smaill for the proofs in this appendix.

Finally, as discussed in *chapter 2* determining whether or not a specification *does* unambiguously specify a procedure is a problem in it's own right (*refinements capture*). Two programs may satisfy the same full specification and yet compute different algorithms (this may be the case with the source and target programs of the specialization pruning transformations (*chapter 5*). Hence it proves useful to show that two of the most common induction schemas are *logically equivalent* (or *inter-derivable*). This guarantees that any two proofs satisfying the same complete specification but differing only in which of the two schemas employed are *functionally equivalent.*

Deriving stepwise induction from course-of-values is fairly easy – this is, in effect, what we do during the *meta-level* transformation of the *Fibonnacci* function (since the *meta-level* transformations are meaning preserving *providing* the source and target proofs share the same *full* specification - *cf. Chapter 6*). We shall now show that course of values induction:

$$\frac{\vdash \forall x : pnat.\ P(x)}{\vdash P(0) \qquad \forall x : pnat,\ \forall y : pnat.\ ((y < x) \rightarrow P(y)) \vdash P(x)}\ ,$$

can be *constructively derived* from normal stepwise induction over the natural numbers

First we give the standard induction over the natural numbers thus:

$$I(P):\ \ (P(0) \wedge \forall n(P(n) \rightarrow P(s(n)))) \rightarrow \forall n P(n)$$

where P is some property (or predicate) of the natural numbers, perhaps containing other free variables. The corresponding course-of-values induction schema is as follows:

$$CV(P):\ \ \forall n(\forall m(m < n \rightarrow P(m)) \rightarrow P(n)) \rightarrow \forall n P(n).$$

We shall now show how $CV(P)$ can be derived from the stepwise recursion schema. $CV(P)$ is reformulated as $CV(P)'$ thus:

$$CV(P)':\ \ (P(0) \wedge \forall n(\forall m(m \leq n \rightarrow P(m)) \rightarrow P(s(n))) \rightarrow \forall n\ P(n).$$

This comes from considering cases $n = 0$ and $n = s(p)$ in the antecedent of $CV(P)$.

It is clear that $CV(P)' \Rightarrow I(P)$, since

$$\forall m(m \leq n \rightarrow P(m)) \rightarrow P(n).$$

For the converse, introduce a new predicate Q defined by

$$Q(n) \leftrightarrow \forall m(m \leq n \rightarrow P(m)).$$

We assume $I(Q)$ and show $CV(P)'$. Suppose

$$P(0) \wedge \forall n(\forall m(m \leq n \rightarrow P(m)) \rightarrow P(s(n)))$$

and show $\forall n\, P(n)$ as follows.

$$\begin{aligned} Q(0) &\leftrightarrow \forall m(m \leq 0 \rightarrow P(m)) \\ &\leftrightarrow P(0), \end{aligned}$$

and

$$\begin{aligned} & Q(n) \rightarrow Q(s(n)) \\ \Leftrightarrow\ & \forall m(m \leq n \rightarrow P(m)) \rightarrow \forall m(m \leq s(n) \rightarrow P(m)) \\ \Leftrightarrow\ & \forall m(m \leq n \rightarrow P(m)) \rightarrow (\forall m(m \leq n \rightarrow P(m)) \wedge P(s(n))) \\ \Leftrightarrow\ & \forall m(m \leq n \rightarrow P(m)) \rightarrow P(s(n)). \end{aligned}$$

Thus, using the two conjuncts of our assumption to infer $Q(0)$ and $\forall n(Q(n) \rightarrow Q(s(n)))$ respectively we conclude

$$Q(0) \wedge \forall n(Q(n) \rightarrow Q(s(n)))$$

and so by $I(Q)$

$$\forall n Q(n).$$

But $Q(n) \rightarrow P(n)$ so

$$\forall n\, P(n). \quad \square$$

Appendix B

Sample Output Runs of the omts

In this appendix we illustrate the OMTS at work by showing an edited selection of OMTS proof transformations as they appear, online, to the user (we omit many of the lengthy stages corresponding to type check mapping). For reasons of brevity, we select a representative example for each of the main OMTS transformation categories:

1. PSS specialization (normalization and dependency pruning),

2. PSS specialization (induction grounding),

3. an example of transformations on an incompletely (weakly) specified proof (the specialization of a weakly specified sorting algorithm),

4. and an example of IPOS recursive program optimization through induction transformation and *proof tupling* (the optimization of an exponential procedure, synthesized through *course_of_values* induction, to a linear procedure (automatically) synthesized through *stepwise* induction).

All transformations are automatic, with the user specifying the name of the source proof, and which particular application of the OMTS he/she desires. All examples, except the induction grounding on *insertion sort* are transformations on *completely* specified proofs thus ensuring source-to-target correctness § 2.2.9. Transformations

on under-specified proofs follow essentially the same course, except there are generally far less verification steps involved due to the relative lack of specification content.

In each example we only display the resulting target proof. However, the sample runs illustrate the relevant portions of the source since in each case portions of the source proofs, the complete source rule-trees, and sub-lists there of, are displayed.

At certain points, comments are added to the system output to enhance readability.

In each case, the OMTS is called by the `omts` prompt.

(1) *Partial Evaluation :* Specialization on function argument

First we show the PSS at work on the *upper bound* example, § 4.3.4. The chosen input parameter to be initialized (instantiated) is y (although x could also have been chosen, with the consequence that no dependency pruning would be possible – § 4.3.5). The chosen initialization for y is 0 (although the user is free to choose *any* desired intialization).

The PSS is fairly flexible such that we have the choice of whether or not to attempt dependency pruning following normalization. The system tells the user whether or not dependency pruning is a possible option (i.e. whether or not the post conditions, § 4.3.4, for dependency pruning are satisfied). The user can then choose whether to interupt the process to view the normalized proof, rule tree and extract, or whether to continue straight away with the dependency pruning. For the benefit of the reader, we choose the former path to display maximum information.

```
| ?- omts.

SPECIALIZATION or OPTIMIZATION (answer s or o) ? s.

OMTS SPECIALIZATION (PSS) ACTIVATED.

COMPLETE OR INCOMPLETE TRANS (answer c or i) ? c.

This transformation is correctness guaranteed:
upon termination, source and target extracts
will satisfy the same *complete* specification.

chatty setting (answer on or off) ? on.
chatty setting on

TYPE OF SPECIALIZATION REQUIRED ?
i.e; induction grounding or norm/dep pruning (ig or n_d_p resp.):

|: n_d_p.

NAME OF SOURCE THEOREM: upper_bound.

VARIABLE (ARG.) TO BE EVALUATED DURING SPECIALIZATION: y.

SUBSTITUTED VALUE FOR y (DESIRED INIYIALIZATION): 0.

SOURCE MAIN GOAL:

uppper_bound: [] partial autotactic(idtac)
==> x:int=>y:int=>z:int#x*y<z#x+y<z
by autotactic(idtac)                          %% The root node of
                                              %% the source proof
  [1] complete
  ==> x:int=>y:int=>z:int#x*y<z#x+y<z         %% The Upper Bound thm.
                                              %% terms of the form x:int
                                              %% signify that x is quantified.
INITIALIZED (partially evaluated) MAIN GOAL:

initialized_target : [] complete autotactic(idtac)
==> x:int=>z:int#x*0<z#x+0<z
by autotactic(idtac)              %% The root node of the partially
                                  %% evaluated (initialized) proof
  [1] complete                    %% (with y set to 0).
  ==> x:int=>z:int#x*0<z#x+0<z
```

```
NORMALIZATION OF upper_bound IN PROGRESS...

nested case condition is  y<1 or y<1=>void  %% system locates 2 case splits
outermost case split is  x<1 or x<1=>void

* partially evaluated nested case condition is TRUE: 0<1  %% The normalized
                                                         %% condition

NORMALIZATION WILL CONSIST OF REPLACING CASE-SPLIT y<1 or y<1=>void
 WITH TRUE CASE: y<1
                          %% i.e. the case y<1=>void is redundant when
                          %% y=0

* normalized target rule tree completed...
* applying normalized rule tree to target specification...

*** NORMALIZATION DONE: CASE-SPLIT y<1 or y<1=>void
                       HAS BEEN REPLACED BY seq(0<1,new[v1])

I.E. NORMALIZATION CONDITION IS: v1:0<1

            ****************************************

THERE IS A FURTHER CASE-SPLIT: x<1, ATTEMPT DEPENDANCY PRUNING ? no.

                                        %% We pospone the dependancy
                                        %% so as to examine the
                                        %% normalized proof (optional).
OK, DEPENDANCY PRUNING NOT YET ACTIVATED.

DISPLAY NORMALIZED PROOF ? yes.

target: [] complete autotactic(idtac)
==> x:int=>z:int#x*0<z#x+0<z
by autotactic(idtac)

  [1] complete
  ==> x:int=>z:int#x*0<z#x+0<z
  by intro                      %% Quantifier stripping.

    [1] complete
    1. x:int
    ==> z:int#x*0<z#x+0<z
    by decide(x<1)              %% Case analysis (x<1 or x<1=>void)
```

```
[1] complete
2. v0:x<1
==> z:int#x*0<z#x+0<z
by intro(0+1)                %% Witness a value for z (upper bound)
                             %% when x<1 (by hypothesis v0)
  [1] complete
  ==> 0+1 in int
  by repeat intro            %% Well-formedness check

  [2] complete
  ==> x*0<0+1#x+0<0+1
  by lemma(lemma1)           %% Verification proceeds using lemma1

    [1] complete
    3. v1:(x:int=>
             y:int=>x<1=>x*y<y+1#x+y<y+1)  %% v1 labels lemma1
    ==> x*0<0+1#x+0<0+1
    by elim(v1,on(x))        %% Quantifier stripping on v1

      [1] complete
      ==> x in int
      by repeat intro

      [2] complete
      4. v2:y:int=>x<1=>x*y<y+1#x+y<y+1
      ==> x*0<0+1#x+0<0+1
      by elim(v2,on(0))      %% y in v2 is instantiated to 0
                             %% yielding v3 below
        [1] complete
        ==> 0 in int
        by repeat intro

        [2] complete
        5. v3:x<1=>x*0<0+1#x+0<0+1
        ==> x*0<0+1#x+0<0+1
        by elim(v3)          %% Appeal to v3 to verify goal, thus
                             %% producing below two sub-goals
          [1] complete
          ==> x<1
          by hyp(v0)         %% Goal unifies with hypothesis vo

          [2] complete
          6. v4:x*0<0+1#x+0<0+1
          ==> x*0<0+1#x+0<0+1
          by hyp(v4)         %% Goal unifies with hypothesis v4
```

```
  [3] complete
  3. z:int
  ==> (x*0<z#x+0<z)in u(1)  %% Type checking (universe provision)
  by repeat intro

[2] complete
2. v0:x<1=>void
==> z:int#x*0<z#x+0<z
by seq(0<1,new[v1])       %% Normalized condition, 0<1, is
                          %% sequenced in place of innermost case
  [1] complete
  ==> 0<1
  by arith                %% Arithmetical check

  [2] complete
  3. v1:0<1
  ==> z:int#x*0<z#x+0<z
  by intro(x+1)           %% Witness a value for z (Upper Bound)
                          %% when y=0. remainder of proof follows
    [1] complete          %% similar procedure as before
    ==> x+1 in int
    by repeat intro

    [2] complete
    ==> x*0<x+1#x+0<x+1
    by lemma(lemma2)

      [1] complete
      4. v2:(x:int=>
               y:int=>
                 y<1=>x*y<x+1#x+y<x+1)
      ==> x*0<x+1#x+0<x+1
      by elim(v2,on(x))

        [1] complete
        ==> x in int
        by repeat intro

        [2] complete
        5. v3:(y:int=>
                 y<1=>x*y<x+1#x+y<x+1)
        ==> x*0<x+1#x+0<x+1
        by elim(v3,on(0))
```

```
            [1] complete
            ==> 0 in int
            by repeat intro

            [2] complete
            6. v4:0<1=>x*0<x+1#x+0<x+1
            ==> x*0<x+1#x+0<x+1
            by elim(v4)

              [1] complete
              ==> 0<1
              by hyp(v1)

              [2] complete
              7. v5:x*0<x+1#x+0<x+1
              ==> x*0<x+1#x+0<x+1
              by hyp(v5)

        [3] complete
        4. z:int
        ==> (x*0<z#x+0<z)in u(1)
        by repeat intro

    [2] complete
    ==> int in u(1)
    by repeat intro

EXTRACT ? yes.

lambda(x,
  less(
  x,
  1,
  ((0+1)&term_of(lemma1)of x of 0 of axiom),
  (lambda(v1,(x+1)&term_of(lemma2)of x of 0 of v1)
    of lambda(_,axiom))))

FURTHER OYSTER EVALUATION OF EXTRACT ? yes.  %% Here we simply see if
                                             %% Oyster can further
                                             %% the extract.

no further (partial) evaluation of extract possible.

LAST CHANCE: ATTEMPT DEPENDANCY PRUNING? yes. %% System instructed to
                                              %% proceed with dependancy
```

```
                                        %% pruning.
OK...

case x<1 corresponds to hypothesis new[v0]

DEPENDANCY PRUNING COMPLETED.

INITIALIZED AND NORMALIZED RULE TREE IS:

apply(intro)then[apply(decide(x<1))then[apply(intro(0+1))then[apply(repeat
intro),apply(lemma(lemma1))then[apply(elim(v1,on(x)))then[apply(repeat
intro),apply(e lim(v2,on(0)))then[apply(repeat
intro),apply(elim(v3))then[apply(hyp(v0)),apply( hyp(v4))]]]],apply(repeat
intro)],apply(seq(0<1)then[cos,apply(intro(x+1))then[a pply(repeat
intro),apply(lemma(lemma2))then[apply(elim(v2,on(x)))then[apply(repe at
intro),apply(elim(v3,on(0)))then[apply(repeat
intro),apply(elim(v4))then[appl y(hyp(v1)),apply(hyp(v5))]]]],apply(repeat
intro)]])],apply(repeat intro)]

DEPENDANCY PRUNED RULE TREE IS:

apply(intro)then[apply(idtac)then[apply(idtac),apply(seq(0<1)then
[cos,apply(intro(x+1))then[apply(repeat
intro),apply(lemma(lemma2))then[apply(elim(v1,on(x)))then[apply(repeat
intro),apply(elim(v2,on(0)))then[apply(repeat
intro),apply(elim(v3))then[apply(hyp(v1)),apply(hyp(v4))]]]],apply(repeat
intro)]])],apply(repeat intro)]

DISPLAY TARGET PROOF TREE? yes.

dep_prune_target : [] complete autotactic(idtac)
==> x:int=>z:int#x*0<z#x+0<z
by autotactic(idtac)

  [1] complete
  ==> x:int=>z:int#x*0<z#x+0<z
  by intro

    [1] complete
    1. x:int
    ==> z:int#x*0<z#x+0<z
    by seq(0<1,new[v0])

      [1] complete
      ==> 0<1
```

```
by arith

[2] complete
2. v0:0<1
==> z:int#x*0<z#x+0<z
by intro(x+1)

  [1] complete
  ==> x+1 in int
  by repeat intro

  [2] complete
  ==> x*0<x+1#x+0<x+1
  by lemma(lemma2)

    [1] complete
    3. v1:(x:int=>
             y:int=>y<1=>x*y<x+1#x+y<x+1)
    ==> x*0<x+1#x+0<x+1
    by elim(v1,on(x))

      [1] complete
      ==> x in int
      by repeat intro

      [2] complete
      4. v2:y:int=>y<1=>x*y<x+1#x+y<x+1
      ==> x*0<x+1#x+0<x+1
      by elim(v2,on(0))

        [1] complete
        ==> 0 in int
        by repeat intro

        [2] complete
        5. v3:0<1=>x*0<x+1#x+0<x+1
        ==> x*0<x+1#x+0<x+1
        by elim(v3)

          [1] complete
          ==> 0<1
          by hyp(v0)      %% Here we appeal to normalized condition
                          %% rather than the cases (x<1 or x>1) of
                          %% the normalized proof.
          [2] complete
```

```
            6. v4:x*0<x+1#x+0<x+1
            ==> x*0<x+1#x+0<x+1
            by hyp(v4)

        [3] complete
        3. z:int
        ==> (x*0<z#x+0<z)in u(1)
        by repeat intro

    [2] complete
    ==> int in u(1)
    by repeat intro

EXTRACT? yes.

lambda(x,                                                  %% The target
  (lambda(~,(x+1)&term_of(lemma2)of x of 0 of axiom) %% algorithm
    of lambda(_,axiom)))

FURTHER OYSTER EVALUATION OF EXTRACT? yes.

lambda(x,(x+1) of x of 0 of axiom)
```

---

(2) *Induction Grounding (by Partial Evaluation):* Specialization on length of list

This example illustrates induction grounding on the completely specified *sumlist* proof. The system specializes the length of the input list to the user-desired parameter value (in this case 3 is chosen).

```
SPECIALIZATION or OPTIMIZATION (answer s or o) ? s.

OMTS SPECIALIZATION (PSS) ACTIVATED.
```

```
COMPLETE OR INCOMPLETE TRANS (answer c or i) ? c.

This transformation is correctness guaranteed:
upon termination, target extract
will satisfy a *complete* specialized
specification.

chatty setting (answer on or off) ? on.
chatty setting on

TYPE OF SPECIALIZATION REQUIRED ?
i.e; induction grounding or norm/dep pruning (ig or n_d_p resp.):

|: ig.                          %% Induction grounding selected

NAME OF SOURCE THEOREM: sumlist.

DESIRED SPECIALIZED LENGTH ? 3.

SOURCE SPECIFICATION IS: []==>l:pnat list=>{y:pnat\sumlist(l,y)}.

SPECIALIZED (TARGET) SPECIFICATION IS:

            []==>a:pnat=>b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}.

RULE-TREE FOR SOURCE sumlist:

I = autotactic(idtac)then[intro then[_1732 then[intro(0),  %% Source
intro(explicit(v2+v0))then[repeat intro]],repeat intro]]   %% rule tree

ENTERING SPECIALIZED SPECIFICATION AS MAIN GOAL.

PERFORMING 3*intro APPLICATIONS...done:

sumlist_target : [1,1,1,1] incomplete autotactic(idtac)
1. a:pnat
2. b:pnat                                   %% Target node following
3. c:pnat                                   %% quantifier stripping
==> {y:pnat\sumlist([a::b::c],y)}
by _

PERFORMING  list INDUCTION GROUNDING...

SOURCE BASE CASE WITNESS: intro(0)
```

```
CORRESPONDING TARGET NODE (``base case'' of target grounded ind):

sumlist_target : [1,1,1,1,1,1,1] complete autotactic(idtac)
1. a:pnat
2. b:pnat
3. c:pnat
==> {y:pnat\sumlist(nil,y)}
by intro(0)

PNAT SEQUENCED INTO TARGET CORRESPONDING
TO EACH UNRAVELING OF SOURCE INDUCTION  (FOR length = 3):

sumlist_target : [1,1,1,1] partial autotactic(idtac)
1. a:pnat
2. b:pnat                                  %% Example of induction
3. c:pnat                                  %% grounding sequencing
==> {y:pnat\sumlist([a::b::c],y)}          %% step (in this case for
by seq({y:sumlist([b::c],a1)},new[v2])     %% the tail of the input list).

  [1] partial
  ==> {a1:sumlist([b::c],a1)}              %% Sequencing step produces
                                           %% two sub goals.
  [2] incomplete
  4. v2:{y:sumlist([b::c],y)}
  ==> {y:pnat\sumlist([a::b::c],y)}

SOURCE INDUCTION STEP WITNESS: intro(explicit(v2+v0)) %% Example witness
                                                      %% (for sub-goal [2]
                                                      %% above).
EXAMPLE MAP OF UNFOLDING OF SOURCE INDUCTION STEP:

sumlist_target : [1,1,1,1,1,1,2] complete autotactic(idtac)
1. a:pnat
2. b:pnat
3. c:pnat                                  %% Example node corresponding to
4. v2:pnat                                 %% induction grounding witnessing
==> {y:pnat\sumlist([c],y)}                %% step.
by intro(explicit(v2+c))then repeat intro

** INDUCTION GROUNDING OF sumlist COMPLETED:

sumlist_target : [] complete autotactic(idtac)
==> a:pnat=>b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}
by autotactic(idtac)
```

```
                                        %% Root node of completed (grounded)
  [1] complete                          %% source proof.
  ==> a:pnat=>b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}

RULE-TREE FOR TARGET sumlist:

0 = autotactic(idtac)then[intro then[intro then[intro then
[seq(int,new[v2])then[seq(int,new[v2])then[seq(int,new
[v2])then[intro(0),intro(explicit(v2+c))then repeat intro],
intro(explicit(v2+b))then repeat intro],intro(explicit(v2+a))
then repeat intro],repeat intro],repeat intro],repeat intro]]

DISPLAY TARGET PROOF TREE? yes.

sumlist_target : [] complete autotactic(idtac)
==> a:pnat=>b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}
by autotactic(idtac)

  [1] complete
  ==> a:pnat=>b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}
  by intro

    [1] complete
    1. a:pnat
    ==> b:pnat=>c:pnat=>{y:pnat\sumlist([a::b::c],y)}
    by intro

      [1] complete
      2. b:pnat
      ==> c:pnat=>{y:pnat\sumlist([a::b::c],y)}
      by intro

        [1] complete
        3. c:pnat
        ==> {y:pnat\sumlist([a::b::c],y)}
        by seq({y:pnat\sumlist([b::c],y)},new[v2])

          [1] complete
          ==> {y:pnat\sumlist([b::c],y)}
          by seq({y:pnat\sumlist([c],y)},new[v2])

            [1] complete
            ==> {y:pnat\sumlist([c],y)}
            by seq({y:pnat\sumlist(nil,y)},new[v2])
```

```
            [1] complete
            ==> {y:pnat\sumlist(nil,y)}
            by intro(0) then apply_lemma(sumlist_lemma1)

            [2] complete
            4. v2:pnat
            ==> {y:pnat\sumlist([c],y)}
            by intro(explicit(v2+c))then apply_lemma(sumlist_lemma2)

          [2] complete
          4. v2:pnat
          ==> {y:pnat\sumlist([b::c],y)}
          by intro(explicit(v2+b))then apply_lemma(sumlist_lemma2)

        [2] complete
        4. v2:pnat
        ==> {y:pnat\sumlist([a::b::c],y)}
        by intro(explicit(v2+a))then apply_lemma(sumlist_lemma2)

      [2] complete
      ==> pnat in u(1)
      by repeat intro

    [2] complete
    ==> pnat in u(1)
    by repeat intro

  [2] complete
  ==> pnat in u(1)
  by repeat intro

EXTRACT? yes.

lambda(a,
  lambda(b,
    lambda(c,
      (lambda(v2,v2+a)of (lambda(v2,v2+b)of (lambda(v2,v2+c)of 0))))))

FURTHER OYSTER EVALUATION OF EXTRACT? yes.

lambda(a,lambda(b,lambda(c,c+b+a)))
yes
```

(3) *Induction Grounding (by Partial Evaluation:* Specialization on Weakly Specified Proof with Nested Inductions

This example illustrates two features of the OMTS:

- transformations on proofs from incomplete specifications: the weak specification for the *insertion sort* proof, *insort*, merely specifies the typing restrictions on the (source) programs input and output (*cf.* § 4.3.3 for a discussion regarding specializations on proofs from incomplete – or under-specified – specifications).

- how the system is capable of adapting algorithms containing *nested* recursion schemata through the pruning of *nested* induction schemata. In this example the nested recursion is due to the nested (auxiliary) *insert* function. The system proceeds by first induction grounding the outermost (list) induction, and then proceeding to ground the nested induction (where the instantiation of the initialized parameter – the length of the input list – is the *same* for both inductions). Induction grounding on nested inductions simply recursively invokes the induction grounding mechanism. Hence, to avoid excess repetition, we omit the online details of induction grounding the auxiliary *insert* function, although, for the record, we do show the resulting extract terms. We also ommit the comments since these would merely be repetitive.

```
SPECIALIZATION or OPTIMIZATION (answer s or o) ? s.

OMTS SPECIALIZATION (PSS) ACTIVATED.

COMPLETE OR INCOMPLETE TRANS (answer c or i) ? i.

This transformation is NOT correctness guaranteed:
upon termination, source and target extracts
will satisfy the same *INcomplete* specification.
```

```
chatty setting (answer on or off) ? on.
chatty setting on

TYPE OF SPECIALIZATION REQUIRED ?
i.e; induction grounding or norm/dep pruning (ig or n_d_p resp.):

|: ig.

NAME OF SOURCE THEOREM: insort.

DESIRED SPECIALIZED LENGTH ? 3.

SOURCE SPECIFICATION IS: []==>l:pnat list=>pnat list

          ** SPECIFICATION IS WEAK **

RULE-TREE FOR SOURCE insort:

autotactic(idtac)then[intro then[_1066 then[intro(nil)
then[repeat intro],intro(explicit(term_of(insert)of v0
of v2))then[simplify then repeat intro]],repeat intro]]

SPECIALIZED (TARGET) SPECIFICATION IS:

           []==>a:pnat=>b:pnat=>c:pnat=>pnat list

ENTERING SPECIALIZED SPECIFICATION AS MAIN GOAL.

PERFORMING 3*intro APPLICATIONS...done:

insort_target : [1,1,1,1] incomplete autotactic(idtac)
1. a:pnat
2. b:pnat
3. c:pnat
==> pnat list
by _

PERFORMING  list INDUCTION GROUNDING...

SOURCE BASE CASE WITNESS: intro(nil)

CORRESPONDING TARGET NODE (``base case'' of target grounded ind):

insort_target : [1,1,1,1,1,1,1] complete autotactic(idtac)
```

```
1. a:pnat
2. b:pnat
3. c:pnat
==> pnat list
by intro(nil)then repeat intro

PNAT SEQUENCED INTO TARGET CORRESPONDING
TO EACH UNRAVELING OF SOURCE INDUCTION  (FOR length = 3):

insort_target : [1,1,1,1] partial autotactic(idtac)
1. a:pnat
2. b:pnat
3. c:pnat
==> pnat list
by seq(pnat list,new[v2])

  [1] partial
  ==> pnat list

  [2] incomplete
  4. v2:pnat list
  ==> pnat list

SOURCE STEP CASE WITNESS: intro(explicit(term_of(insert)of v0 of v2))

EXAMPLE MAP OF UNFOLDING OF SOURCE INDUCTION STEP:

insort_target : [1,1,1,1,1,1,2] complete autotactic(idtac)
1. a:pnat
2. b:pnat
3. c:pnat
4. v2:pnat list
==> pnat list
by (intro(explicit(term_of(insert)of c of v2))
    then (simplify then repeat intro))

** INDUCTION GROUNDING OF insort COMPLETED:

insort_target : [] complete autotactic(idtac)
==> a:pnat=>b:pnat=>c:pnat=>pnat list
by autotactic(idtac)

  [1] complete
  ==> a:pnat=>b:pnat=>c:pnat=>pnat list
```

```
RULE-TREE FOR TARGET insort:

0 = autotactic(idtac)then[intro then[intro then[intro th
en[seq(int list,new[v2])then[seq(int list,new[v2])then[s
eq(int list,new[v2])then[intro(nil)then repeat intro,int
ro(explicit(term_of(insert)of c of v2))then simplify the
n repeat intro],intro(explicit(term_of(insert)of b of v2
))then simplify then rep

DISPLAY TARGET PROOF TREE? yes.
insort_target : [] complete autotactic(idtac)
==> a:pnat=>b:pnat=>c:pnat=>pnat list
by autotactic(idtac)

  [1] complete
  ==> a:pnat=>b:pnat=>c:pnat=>pnat list
  by intro

    [1] complete
    1. a:pnat
    ==> b:pnat=>c:pnat=>pnat list
    by intro

      [1] complete
      2. b:pnat
      ==> c:pnat=>pnat list
      by intro

        [1] complete
        3. c:pnat
        ==> pnat list
        by seq(pnat list,new[v2])

          [1] complete
          ==> pnat list
          by seq(pnat list,new[v2])

            [1] complete
            ==> pnat list
            by seq(pnat list,new[v2])

              [1] complete
              ==> pnat list
              by intro(nil)then repeat intro
```

```
            [2] complete
            4. v2:pnat list
            ==> pnat list
            by (intro(explicit(term_of(insert)of c of v2))
                then (simplify then repeat intro))

          [2] complete
          4. v2:pnat list
          ==> pnat list
          by (intro(explicit(term_of(insert)of b of v2))
              then (simplify then repeat intro))

        [2] complete
        4. v2:pnat list
        ==> pnat list
        by (intro(explicit(term_of(insert)of a of v2))
            then (simplify then repeat intro))

      [2] complete
      ==> pnat in u(1)
      by repeat intro

    [2] complete
    ==> pnat in u(1)
    by repeat intro

  [2] complete
  ==> pnat in u(1)
  by repeat intro

EXTRACT? yes.

lambda(a,
  lambda(b,
    lambda(c,
      (lambda(v2,term_of(insert)of a of v2)
        of (lambda(v2,term_of(insert)of b of v2)
             of (lambda(v2,term_of(insert)of c of v2)of nil))))))

FURTHER OYSTER EVALUATION OF EXTRACT? yes.

lambda(a,lambda(b,lambda(c,list_ind(lambda(v2,term_of(insert)of b of v2)of
(lambda(v2,term_of(insert)of c of v2)of
nil),a::nil,[v0,v1,v2,less(v0,a,v0::v2,a::v0::v1)]))))
```

```
THERE IS AN INDUCTIVE AUXILIARY FUNCTION: $insert$
PERFORM NESTED INDUCTION GROUNDING ? yes.

OK. proceeding with nested induction grounding (initialized length = 3).

NESTED SOURCE SPECIFICATION IS: []==> n:int=>l:int list=>int list

NESTED SPECIALIZED (TARGET) SPECIFICATION IS:
                                []==> n:int=>a:int=>b:int=>int list

ENTERING SPECIALIZED NESTED SPECIFICATION AS MAIN GOAL:

insert_target: [] incomplete autotactic(idtac)
==> n:int=>a:int=>b:int=>c:int=>int list
by _
```

.....DETAILS OMMITED.....

```
** GROUNDING OF NESTED INDUCTIVE PROOF COMPLETE.

insert_target: [] complete autotactic(idtac)
==> n:int=>a:int=>b:int=>c:int=>int list
by intro

  [1] complete
  1. n:int
  ==> a:int=>b:int=>c:int=>int list

  [2] complete
  ==> int in u(1)

DISPLAY PROOF ? no.

GROUNDED NESTED EXTRACT? yes.

lambda(n,
  lambda(a,
    lambda(b,
      lambda(c,
        (lambda(v2,less(c,n,c::v2,n::c::nil))
          of (lambda(v2,less(b,n,b::v2,n::b::c::nil))
               of (lambda(v2,less(a,n,a::v2,n::a::b::c::nil))
```

```
                    of (n::nil))))))))

FURTHER OYSTER EVALUATION OF EXTRACT? yes.

lambda(n,
  lambda(a,
    lambda(b,
      lambda(c,less(c,n,c::lambda(v2,less(b,n,b::v2,n::b::c::nil))
       of (lambda(v2,less(a,n,a::v2,n::a::b::c::nil))
         of (n::nil)),n::c::nil)))))

GROUNDED insort EXTRACT? yes.

lambda(a,
  lambda(b,
    lambda(c,
      (lambda(v2,term_of(insert_target)of a of v2)
        of (lambda(v2,term_of(insert_target)of b of v2)
             of (lambda(v2,term_of(insert_target)of c of v2)of nil))))))
```

---

(4) *Induction Transformation:* Optimization of course of values to stepwise recursion

This example illustrates

- the exponential (*course_of_values*) to linear (*stepwise*) transformation of the *Fibonacci* function, by the IPOS.
- the IPOS *proof tupling* at work.

Both source and target proofs satisfy the same complete specification and the optimization is hence source-to-target correctness guaranteed.

```
| ?- omts.

SPECIALIZATION or OPTIMIZATION (answer s or o) ? o.

OMTS OPTIMIZATION (IPOS) ACTIVATED.

COMPLETE OR INCOMPLETE TRANS (answer c or i) ? c.

This transformation is correctness guaranteed:
upon termination, source and target extracts
will satisfy the same *complete* specification.

chatty setting (answer on or off) ? on.
chatty setting on

NAME OF SOURCE PROOF ? fibonacci.

TRANSFORMATION IN PROGRESS....

SOURCE *AND* TARGET SPEC: []==>x:pnat=>{y:pnat\fib_rel(x,y)}

LEMMAS:                                  %% System locates, and displays, all
       fib_lemma1: fib_rel(0,s(0))       %% lemmas used in source proof.
       fib_lemma2: fib_rel(s(0),s(0))
       fib_lemma3: x:pnat=>y:pnat=>z:pnat=>s(0)<x=>(fib_rel(pred(pred(x)),y)#
                                    fib_rel(pred(x),z))=>fib_rel(x,y+z)

RULE TREE FOR fibonacci SOURCE FORMED.
 i.e OYSTER PROOF REP PROCESSED INTO SEMANTIC UNITS:

UNIT CORRESPONDING TO SOURCE INDUCTION:  %% i.e. that portion of rule-tree
                                         %% corresponding to induction sub-
                                         %% proof.

decide(v2=0 in pnat)then[rewrite(v1)then[intro(s(0))then
[lemma(fib_lemma1,_19671)]], decide(v2=s(0)in pnat)then[rewrite(v3)then
[intro(s(0))then[lemma(fib_lemma2,_20089)]],elim(v0,on(pred(v2)))then
[arith,elim(v0,on(pred(pred(v2))))then[arith,elim(v4)then[elim(v5)then
[lemma(fib_lemma3)then[elim_on(v12,[v2,v5,v4],v15)then[thin([v12,v13,v14])then
[intro(plus(v5,v4))then[elim(v15)then[arith,tautology(elim(v12)then[intro
then[intro,intro],intro])]]]]]]]]]]

************************************************
CASE ANALYSIS: v2=0 in pnat       %% We are dealing with course_of_values
                                  %% induction, so system locates the
```

```
                                %% induction cases by locating the
                                %% explicit applications of the decide
                                %% rule.

RECORDED BASE AND/OR INDUCTION STEP:
rewrite(v1)then[intro(s(0))then[lemma(fib_lemma1,_19671)]] %% rule-tree unit
                                                          %% for 1st base.

***********************************************

CASE ANALYSIS: v2=s(0)in pnat

RECORDED BASE AND/OR INDUCTION STEP:
rewrite(v3)then[intro(s(0))then[lemma(fib_lemma2,_20089)]] %% rule-tree unit
                                                          %% for 1st base.

RECORDED BASE AND/OR INDUCTION STEP:                      %% rule-tree unit
                                                          %% for step.

elim(v0,on(pred(v2)))then[arith,elim(v0,on(pred(pred(v2))))then
[arith,elim(v4)then[elim(v5)then[lemma(fib_lemma3)then
[elim_on(v12,[v2,v5,v4],v15)then[thin([v12,v13,v14])then
[intro(plus(v5,v4))then[elim(v15)then[arith,tautology(elim(v12)then
[intro then[intro,intro],intro])]]]]]]]]]_41699

SUBSIDIARY CALLS INTRODUCED AT SOURCE STEP: [pred(pred(v2)),pred(v2)]

                          %% the system locates and lists the subsidiary
                          %% function calls.

TUPLE LENGTH IS 2         %% Since the maximum number of applications
                          %% of the common generator function, pred, is 2.

recursive args of 2 tuple components: [s(x),x] %% Conversion of subsidiary
                                               %% calls to constructor form.

STRUCTURE TO BE SEQUENCED INTO PROOF AS SUB GOAL:

{a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)} %% Tuple structure to be
                                                 %% sequenced, i.e.:
 SO SEQUENCED SUB-GOAL WILL BE:

seq({a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)},new[tuple])

TARGET TOP-LEVEL GOAL:            %% Identical to source root goal.
```

```
target: [] incomplete autotactic(idtac)
==> x:pnat=>{y:pnat\fib_rel(x,y)}
by _

Performing intro application...done.        %% i.e. Quantifier stripping.

target: [] partial autotactic(idtac)        %% Node following quantifier
==> x:pnat=>{y:pnat\fib_rel(x,y)}           %% stripping.
by intro

  [1] partial
  1. x:pnat
  ==> {y:pnat\fib_rel(x,y)}

  [2] complete
  ==> pnat in u(1)

PERFORMING SEQUENCING STEP...done:          %% Sequencing step produces two
                                            %% subgoals:
1. x:pnat
==> {y:pnat\fib_rel(x,y)}
by seq(
  ({a1:pnat\fib_rel(s(x),a1)}#{y:pnat\fib_rel(x,y)}),
  new[tuple])

  [1] complete
  ==> {a1:pnat\fib_rel(s(x),a1)}#{y:pnat\fib_rel(x,y)}

  [2] complete
  2. tuple:({a1:pnat\fib_rel(s(x),a1)})#
       {y:pnat\fib_rel(x,y)}
  ==> {y:pnat\fib_rel(x,y)}

                                            %% System moves down to sub-goal
                                            %% [2] for induction application.
PERFORMING stepwise INDUCTION ON x:

target: [1,1] complete autotactic(idtac)
1. x:pnat
==> {a1:pnat\fib_rel(s(x),a1)}#{y:pnat\fib_rel(x,y)}
by elim(x,new[ind_obj,step_hyp])                    %% Stepwise induction.

  [1] complete
  ==> {a1:pnat\fib_rel(s(0),a1)}#{y:pnat\fib_rel(0,y)} %% Base case
```

```
  [2] complete
  2. ind_obj:pnat
  3. step_hyp:({a1:pnat\fib_rel(s(ind_obj),a1)})#      %% Induction hypothesis.
       {y:pnat\fib_rel(ind_obj,y)}
  ==> ({a1:pnat\fib_rel(s(s(ind_obj)),a1)})#           %% Step case.
    {y:pnat\fib_rel(s(ind_obj),y)}

TARGET BASE CASE (intro splits tuple into components):

target: [1,1,1] partial autotactic(idtac)
1. x:pnat
==> {a6:pnat\fib_rel(s(0),a6)}#{y:pnat\fib_rel(0,y)}
by intro

  [1] incomplete
  ==> {a6:pnat\fib_rel(s(0),a6)}

  [2] complete
  ==> {y:pnat\fib_rel(0,y)}

TARGET 1st BASE CASE WITNESSING NODE:   %% System locates and uses source
                                        %% base case witnesses so as to
                                        %% provide witnesses for target
                                        %% base tuple components.
                                        %% The system then uses the lemmas
                                        %% to verify the witnessed values.

target: [1,1,1,2] partial autotactic(idtac)
1. x:pnat
==> {y:pnat\fib_rel(0,y)}
by intro(s(0))

  [1] complete
  ==> s(0)in pnat

  [2] incomplete
  ==> fib_rel(0,s(0))

  [3] incomplete
  2. y:pnat
  ==> fib_rel(0,y)in u(1)

VERIFYING 1st BASE CASE WITNESS:
```

```
APPLYING LEMMA apply_lemma(fib_lemma1)

target: [1,1,1,2,2] incomplete autotactic(idtac)
1. x:pnat
==> fib_rel(0,s(0))
by _

APPLIED LEMMA
target: [1,1,1,2,2] complete autotactic(idtac)
1. x:pnat
==> fib_rel(0,s(0))
by (lemma(fib_lemma1,new[v0])
    then (dequantify then elim_on(v0,[],v0)then hyp(_)))

TARGET 2nd BASE CASE WITNESSING NODE:

target: [1,1,1,1] incomplete autotactic(idtac)
1. x:pnat
==> {a6:pnat\fib_rel(s(0),a6)}
by _

target: [1,1,1,1] partial autotactic(idtac)
1. x:pnat
==> {a6:pnat\fib_rel(s(0),a6)}
by intro(s(0))

  [1] complete
  ==> s(0)in pnat

  [2] incomplete
  ==> fib_rel(s(0),s(0))

  [3] incomplete
  2. a6:pnat
  ==> fib_rel(s(0),a6)in u(1)

VERIFYING 2nd BASE CASE WITNESS:

APPLYING LEMMA apply_lemma(fib_lemma2)

target: [1,1,1,1,2] incomplete autotactic(idtac)
1. x:pnat
==> fib_rel(s(0),s(0))
by _
```

```
APPLIED LEMMA

target: [1,1,1,1,2] complete autotactic(idtac)
1. x:pnat
==> fib_rel(s(0),s(0))
by (lemma(fib_lemma2,new[v0])
    then (dequantify then elim_on(v0,[],v0)then hyp(_)))

INDUCTION STEP CASE:                %% For the target induction step, the
                                    %% system uses the (constructor form) of
                                    %% the source induction witnesses.
                                    %% The proof is completed by using the
                                    %% same lemma, lemma3, as in the source.

target: [1,1,2,1] partial autotactic(idtac)
1. x:pnat
2. ind_obj:pnat
3. step_hyp:({a6:pnat\fib_rel(s(ind_obj),a6)})#
     {y:pnat\fib_rel(ind_obj,y)}
4. a8:{a6:pnat\fib_rel(s(ind_obj),a6)}
5. a9:{y:pnat\fib_rel(ind_obj,y)}
6. a10:(step_hyp
        = (a8&a9)
        in ({a6:pnat\fib_rel(s(ind_obj),a6)})#
           ({y:pnat\fib_rel(ind_obj,y)}))
==> ({a6:pnat\fib_rel(s(s(ind_obj)),a6)})#
  {y:pnat\fib_rel(s(ind_obj),y)}
by intro

  [1] incomplete
  ==> {a6:pnat\fib_rel(s(s(ind_obj)),a6)}

  [2] incomplete
  ==> {y:pnat\fib_rel(s(ind_obj),y)}

WITNESSING STEP OF 1st TUPLE COMPONENT:
target: [1,1,2,1,2] complete autotactic(idtac)
1. x:pnat
2. ind_obj:pnat
3. step_hyp:({a6:pnat\fib_rel(s(ind_obj),a6)})#
     {y:pnat\fib_rel(ind_obj,y)}
4. a8:{a6:pnat\fib_rel(s(ind_obj),a6)}
5. a9:{y:pnat\fib_rel(ind_obj,y)}
6. a10:(step_hyp
        = (a8&a9)
```

```
        in ({a6:pnat\fib_rel(s(ind_obj),a6)})#
           ({y:pnat\fib_rel(ind_obj,y)}))
==> {y:pnat\fib_rel(s(ind_obj),y)}
by hyp(a8)

WITNESSING STEP OF 2nd TUPLE COMPONENT:

target: [1,1,2,1,1] complete autotactic(idtac)
1. x:pnat
2. ind_obj:pnat
3. step_hyp:({a6:pnat\fib_rel(s(ind_obj),a6)})#
     {y:pnat\fib_rel(ind_obj,y)}
4. a8:{a6:pnat\fib_rel(s(ind_obj),a6)}
5. a9:{y:pnat\fib_rel(ind_obj,y)}
6. a10:(step_hyp
        = (a8&a9)
        in ({a6:pnat\fib_rel(s(ind_obj),a6)})#
           ({y:pnat\fib_rel(ind_obj,y)}))
==> {a6:pnat\fib_rel(s(s(ind_obj)),a6)}
by intro(a8+a9)

  [1] complete
  ==> a8+a9in pnat

  [2] complete
  ==> fib_rel(s(s(ind_obj)),a8+a9)   %% lemma3 is used to complete this
                                     %% branch (not shown here).
  [3] complete
  7. a6:pnat
  ==> fib_rel(s(s(ind_obj)),a6)in u(1)

**** SEQUENCED GOAL:

      seq({a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)},new[tuple])

HAS BEEN PROVED.

RETURNING TO MAIN GOAL (i.e. other branch of sequencing step):

target: [1,2] partial autotactic(idtac)
1. x:pnat
2. tuple:({a6:pnat\fib_rel(s(x),a6)})#
     {y:pnat\fib_rel(x,y)}
==> {y:pnat\fib_rel(x,y)}
by elim(tuple,new[a11,a12,a13])
```

```
  [1] incomplete
  3. a11:{a6:pnat\fib_rel(s(x),a6)}
  4. a12:{y:pnat\fib_rel(x,y)}
  5. a13:(tuple
          = (a11&a12)
          in ({a6:pnat\fib_rel(s(x),a6)})#
             ({y:pnat\fib_rel(x,y)}))
  ==> {y:pnat\fib_rel(x,y)}

TRANSFERING REQUIRED VALUE, y:pnat, TO MAIN GOAL:

target: [1,2,1] complete autotactic(idtac)
1. x:pnat
2. tuple:({a6:pnat\fib_rel(s(x),a6)})#
     {y:pnat\fib_rel(x,y)}
3. a11:{a6:pnat\fib_rel(s(x),a6)}
4. a12:{y:pnat\fib_rel(x,y)}
5. a13:(tuple
        = (a11&a12)
        in ({a6:pnat\fib_rel(s(x),a6)})#
           ({y:pnat\fib_rel(x,y)}))
==> {y:pnat\fib_rel(x,y)}
by hyp(a12)

** TRANSFORMATION COMPLETE:

target: [] complete autotactic(idtac)
==> x:pnat=>{y:pnat\fib_rel(x,y)}
by intro

  [1] complete
  1. x:pnat
  ==> {y:pnat\fib_rel(x,y)}

  [2] complete
  ==> pnat in u(1)

DISPLAY TARGET PROOF TREE? yes.

target: [] complete autotactic(idtac)
==> x:pnat=>{y:pnat\fib_rel(x,y)}
by intro

  [1] complete
```

```
1. x:pnat
==> {y:pnat\fib_rel(x,y)}
by seq(
  ({a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)}),
  new[tuple])

  [1] complete
  ==> {a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)}
  by elim(x,new[ind_obj,step_hyp])

    [1] complete
    ==> {a6:pnat\fib_rel(s(0),a6)}#{y:pnat\fib_rel(0,y)}
    by intro

      [1] complete
      ==> {a6:pnat\fib_rel(s(0),a6)}
      by intro(s(0))

        [1] complete
        ==> s(0)in pnat
        by repeat intro

        [2] complete
        ==> fib_rel(s(0),s(0))
        by (lemma(fib_lemma2,new[v0])
            then (dequantify then elim_on(v0,[],v0)then hyp(_)))

        [3] complete
        2. a6:pnat
        ==> fib_rel(s(0),a6)in u(1)
        by repeat wfftac

      [2] complete
      ==> {y:pnat\fib_rel(0,y)}
      by intro(s(0))

        [1] complete
        ==> s(0)in pnat
        by repeat intro

        [2] complete
        ==> fib_rel(0,s(0))
        by (lemma(fib_lemma1,new[v0])
            then (dequantify then elim_on(v0,[],v0)then hyp(_)))
```

```
    [3] complete
    2. y:pnat
    ==> fib_rel(0,y)in u(1)
    by repeat wfftac

[2] complete
2. ind_obj:pnat
3. step_hyp:({a6:pnat\fib_rel(s(ind_obj),a6)})#
     {y:pnat\fib_rel(ind_obj,y)}
==> ({a6:pnat\fib_rel(s(s(ind_obj)),a6)})#
  {y:pnat\fib_rel(s(ind_obj),y)}
by elim(step_hyp,new[a8,a9,a10])

  [1] complete
  4. a8:{a6:pnat\fib_rel(s(ind_obj),a6)}
  5. a9:{y:pnat\fib_rel(ind_obj,y)}
  6. a10:(step_hyp
          = (a8&a9)
          in ({a6:pnat\fib_rel(s(ind_obj),a6)})#
             ({y:pnat\fib_rel(ind_obj,y)}))
  ==> ({a6:pnat\fib_rel(s(s(ind_obj)),a6)})#
    {y:pnat\fib_rel(s(ind_obj),y)}
  by intro

    [1] complete
    ==> {a6:pnat\fib_rel(s(s(ind_obj)),a6)}
    by intro(a8+a9)

      [1] complete
      ==> a8+a9in pnat
      by repeat wfftac

      [2] complete
      ==> fib_rel(s(s(ind_obj)),a8+a9)
      by (lemma(fib_lemma3,new[v0])
        then (dequantify then elim_on(v0,[],v0)then hyp(_)))

      [3] complete
      7. a6:pnat
      ==> fib_rel(s(s(ind_obj)),a6)in u(1)
      by repeat wfftac

    [2] complete
    ==> {y:pnat\fib_rel(s(ind_obj),y)}
    by hyp(a8)
```

```
    [2] complete
    2. tuple:({a6:pnat\fib_rel(s(x),a6)})#
         {y:pnat\fib_rel(x,y)}
    ==> {y:pnat\fib_rel(x,y)}
    by elim(tuple,new[a11,a12,a13])

      [1] complete
      3. a11:{a6:pnat\fib_rel(s(x),a6)}
      4. a12:{y:pnat\fib_rel(x,y)}
      5. a13:(tuple
              = (a11&a12)
              in ({a6:pnat\fib_rel(s(x),a6)})#
                 ({y:pnat\fib_rel(x,y)}))
      ==> {y:pnat\fib_rel(x,y)}
      by hyp(a12)

  [2] complete
  ==> pnat in u(1)
  by repeat intro

DISPLAY TARGET EXTRACT... ? yes.
...WITH OR WITHOUT SOURCE EXTRACT (for comparison) ?
answer w or wo: w.

SOURCE EXTRACT:
lambda(x,
  cv_ind(
  x,
  [v2,v0,
   pnat_eq(
   v2,
   0,
   s(0),
   pnat_eq(v2,s(0),s(0),v0 of pred(pred(v2))+v0 of pred(v2)))]))

TARGET EXTRACT:
lambda(x,
  (lambda(tuple,spread(tuple,[~,a12,a12]))
    of p_ind(
       x,
       (s(0)&s(0)),
       [~,step_hyp,spread(step_hyp,[a8,a9,a8+a9&a8])])))

FURTHER OYSTER EVALUATION OF TARGET EXTRACT? yes.
```

```
lambda(x,spread(p_ind(x,s(0)&s(0),[~,step_hyp,spread(step_hyp,[a8,a9,plus(a8,a9)&a8])]),[~,a12,a12]))

Display target rule_tree abstraction? yes.
...WITH OR WITHOUT SOURCE RULE-TREE (for comparison) ?
answer w or wo: w.

SOURCE RULE-TREE:

autotactic(repeat (try intro then wfftac))then[intro then[elim(x,cv)then
[decide(v2=0 in pnat)then[rewrite(v1)then[intro(s(0))then
[lemma(fib_lemma1,_311950)]],decide(v2=s(0)in pnat)then[rewrite(v3)then
[intro(s(0))then[lemma(fib_lemma2,_312368)]],elim(v0,on(pred(v2)))then
[arith,elim(v0,on(pred(pred(v2))))then[arith,elim(v4)then[elim(v5)then
[lemma(fib_lemma3)then[elim_on(v12,[v2,v5,v4],v15)then[thin([v12,v13,v14])then
[intro(plus(v5,v4))then[elim(v15)then[arith,tautology(elim(v12)then[intro
then[intro,intro],intro])]]]]]]]]]]]]]]

TARGET RULE-TREE:

intro
then[seq({a6:pnat\fib_rel(s(x),a6)}#{y:pnat\fib_rel(x,y)},new[tuple])then
[elim(x,new[ind_obj,step_hyp])then[intro then[intro(s(0))then[repeat
intro,lemma(fib_lemma2,new[v0])then dequantify then
elim_on(v0,[],v0)then hyp(_325018),repeat
wfftac],intro(s(0))then[repeat intro,lemma(fib_lemma1,new[v0])then
dequantify then elim_on(v0,[],v0)then hyp(_325159),repeat
wfftac]],elim(step_hyp,new[a8,a9,a10])then[intro
then[intro(plus(a8,a9))then[repeat
wfftac,lemma(fib_lemma3,new[v0])then dequantify then
elim_on(v0,[],v0)then hyp(_325288),repeat
wfftac],hyp(a8)]]],elim(tuple,new[a11,a12,a13])then[hyp(a12)]],repeat
intro]

COMPARISON OF AVERAGE (CPU) RUN TIMES ? yes.

OK...
NUMBER OF RUNS (FOR AVERAGE): 10.

INPUT (INT): 4.

source evaluations...

5 5 5 5 5 5 5 5 5 5
```

```
Average CPU time for source fibonacci over 10 run(s): 13780.0

target evaluations...

5 5 5 5 5 5 5 5 5 5

Average CPU time for target fibonacci over 10 run(s): 5075.0

Averaging over 10 run(s), with 4 as input, the target is 2.71527 more efficient
than the source.

yes
| ?-
```
